Asynchronous Multiparty Session Type Implementability is Decidable – Lessons Learned from Message Sequence Charts

Felix Stutz
MPI-SWS, Kaiserslautern, Germany

Abstract

Multiparty session types (MSTs) provide efficient means to specify and verify asynchronous message-passing systems. For a global type, which specifies all interactions between roles in a system, the implementability problem asks whether there are local specifications for all roles such that their composition is deadlock free and generates precisely the specified executions. Decidability of the implementability problem is an open question. We answer it positively for global types with generalised choice that allow a sender to send to different receivers and a receiver to receive from different senders upon branching. To achieve this, we generalise results from the domain of high-level message sequence charts (HMSCs). This connection also allows us to comprehensively investigate how HMSC techniques can be adapted to the MST setting. This comprises techniques to make the problem algorithmically more tractable as well as a variant of implementability which may open new design space for MSTs. Inspired by potential performance benefits, we introduce a generalisation of the implementability problem that we, unfortunately, prove to be undecidable.

2012 ACM Subject Classification Theory of computation → Concurrency

Keywords and phrases Multiparty session types, Verification, Message sequence charts

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

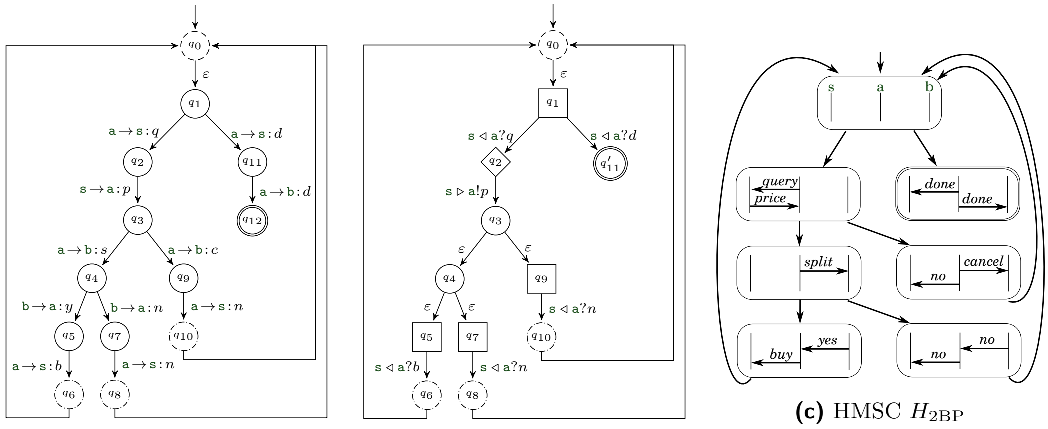
1 Introduction

Distributed message-passing systems are omnipresent and, therefore, designing and implementing them correctly is very important. However, this is a very difficult task at the same time. In fact, it is well-known that the verification problem is algorithmically undecidable in general due to the combination of asynchrony (messages are buffered) and concurrency [15].

Multiparty Session Type (MST) frameworks provide efficient means to specify and verify such distributed message-passing systems. MSTs (and their binary counterpart) are not only of theoretical interest but have been implemented for many mainstream programming languages [6, 54, 62, 58, 74, 70, 25]. They have also been applied to various other domains like operating systems [36], cyber-physical systems [65], timed systems [11], distributed algorithms [57], web services [86], and smart contracts [33]. In MST frameworks, global types are global specifications, which comprise all interactions between roles in a protocol. From a design perspective, it makes sense to start with such a global protocol specification — instead of a system with arbitrary communication between roles and a specification to satisfy.

Let us consider a variant of the well-known two buyer protocol from the MST literature, e.g., [75, Fig. 4 (2)]. Two Buyers `a` and `b` purchase a sequence of items from Seller `s`. We informally describe the protocol and *emphasise* the interactions. At the start and after every purchase (attempt), Buyer `a` can decide whether to buy the next item or whether they are *done*. For each item, Buyer `a` *queries* its price and the Seller `s` replies with the *price*. Subsequently, Buyer `a` decides whether to *cancel* the purchase process for the current item or proposes to *split* to Buyer `b` that can *accept* or *reject*. In both cases, Buyer `a` notifies the Seller `s` whether they want to *buy* the item or *not*. This protocol can be specified with the following global type:

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:43
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



(a) State machine for semantics of $\mathbf{G}_{2\mathrm{BP}}$

(b) Projection of $\mathbf{G}_{2\mathrm{BP}}$ on to s without merge

(c) HMSC $H_{2\mathrm{BP}}$

Figure 1 Two Buyer Protocol: the finite state machine for the semantics of $\mathbf{G}_{2\mathrm{BP}}$ on the left, the first step of projection in the middle, and as HMSC on the right; a transition label $\mathtt{a} \to \mathtt{s} : q$ jointly specifies a send event $\mathtt{a} \triangleright \mathtt{s}!q$ for Buyer a and a receive event $\mathtt{s} \triangleleft \mathtt{a}?q$ for Seller s; styles of states indicate their kind, e.g., recursion states (dashed lines) while final states have double lines

$$\mathbf{G}_{2\mathrm{BP}} \quad := \quad \mu t. + \begin{cases} \mathtt{a} \to \mathtt{s} : \mathit{query}.\, \mathtt{s} \to \mathtt{a} : \mathit{price}. + \begin{cases} \mathtt{a} \to \mathtt{b} : \mathit{split}.\, (\mathtt{b} \to \mathtt{a} : \mathit{yes}.\, \mathtt{a} \to \mathtt{s} : \mathit{buy}.\, t + \mathtt{b} \to \mathtt{a} : \mathit{no}.\, \mathtt{a} \to \mathtt{s} : \mathit{no}.\, t) \\ \mathtt{a} \to \mathtt{b} : \mathit{cancel}.\, \mathtt{a} \to \mathtt{s} : \mathit{no}.\, t \end{cases} \\ \mathtt{a} \to \mathtt{s} : \mathit{done}.\, \mathtt{a} \to \mathtt{b} : \mathit{done}.\, 0 \end{cases} .$$

The first term μt binds the recursion variable t which is used at the end of the first two lines and allows the protocol to recurse back to this point. Subsequently, $+$ and the curly bracket indicate a choice that is taken by Buyer a as it is the sender for the next interaction, e.g., $\mathtt{a} \to \mathtt{s} : \mathit{query}$. For our asynchronous setting, this term jointly specifies the send event $\mathtt{a} \triangleright \mathtt{s}!\mathit{query}$ for Buyer a and its corresponding receive event $\mathtt{s} \triangleleft \mathtt{a}?\mathit{query}$ for Seller s, which may happen with arbitrary delay. The state machine in Figure 1a illustrates its semantics.

The Implementability Problem for Global Types and the MST Approach

A global type provides a global view of the intended protocol. However, when implementing a protocol in a distributed setting, one needs a local specification for each role. The *implementability problem* for a global type asks whether there are local specifications for all roles such that, when complying with their local specifications, their composition never gets stuck and exposes the same executions as specified by the global type. This is a challenging problem because roles can only partially observe the execution of a system: each role only knows the messages it sent and received and, in an asynchronous setting, a role does not know when one of its messages will be received by another role.

In general, one distinguishes between a role in a protocol and the process which implements the local specification of a role in a system. We use the local specifications directly as implementations so the difference is not essential and we use the term role instead of process.

Classical MST frameworks employ a partial *projection operator* with an in-built *merge operator* to solve the implementability problem. For each role, the projection operator takes the global type and removes all interactions the role is not involved in. Figure 1a illustrates the semantics of $\mathbf{G}_{2\mathrm{BP}}$ while Figure 1b gives the projection on to Seller s before the merge operator is applied — in both, messages are abbreviated with their first letter. It is easy to see that this introduces non-determinism, e.g., in q_3 and q_4, which shall be resolved by the merge operator. Most merge operators can resolve the non-determinism in Figure 1b. A merge operator checks whether it is safe to merge the states and it might fail so it is a partial operation. For instance, every kind of state, indicated by a state's style in Figure 1b, can only be merged with states of the same kind and states of circular shape. For a role, the result of the projection, if defined, is a local type. They act as local specifications and their syntax is similar to the one of global types.

Classical projection operators are a best-effort technique. This yields good (mostly linear) worst-case complexity but comes at the price of rejecting implementable global types. Intuitively, classical projection operators consider a limited search space for local types. They bail out early when encountering difficulties and do not unfold recursion. In addition, most MST frameworks do effectively not allow a role to send to different receivers or receive from different senders upon branching. This restriction is called *directed choice* — in contrast to *generalised choice* which allows such patterns. Among the classical projection operators, the one by Majumdar et al. [64] is the only to handle global types with *generalised choice* but suffers from the shortcomings of a classical projection approach. We define different merge operators from the literature and visually explain their supported features by example. We show that the presented classical projection/merge operators fail to project implementable variations of the two buyer protocol. This showcases the sources of incompleteness for the classical projection approach. For non-classical approaches, we refer to Section 7.

As a best-effort technique, it is natural to focus on efficiency rather than completeness. The work by Castagna et al. [19] is a notable exception even though their notion of completeness [19, Def. 4.1] is not as strict as the one considered in this work and only a restricted version of their characterisation is algorithmically checkable. In general, it is not known whether the implementability problem for global types, with directed or generalised choice, is decidable. We answer this open question positively for global types with generalised choice. To this end, we relate the implementability problem for global types with the safe realisability problem for high-level message sequence charts and generalise results for the latter.

Lessons Learned from Message Sequence Charts

The two buyer protocol $\mathbf{G}_{2\mathrm{BP}}$ can also be specified as high-level message sequence chart (HMSC). It is illustrated in Figure 1c. Each block is a basic message sequence chart (BMSC) which intuitively corresponds to straight-line code. In each of those, time flows from top to bottom and each role is represented by a vertical line. We only give the names in the initial block, which is marked by an incoming arrow at the top. An arrow between two role lines specifies sending and receiving a message with the corresponding label. The graph structure adds branching, which corresponds to choice in global types, and control flow. Top branches from the global type are on the left in the HMSC while bottom branches are on the right.

While research on MSTs and HMSCs has been pursued quite independently, the MST literature frequently uses HMSC-like visualisations for global types, e.g., [18, Fig. 1] and [49, Figs. 1 and 2]. The first formal connection was recently established by Stutz and Zufferey [77].

The HMSC approach to the implementability problem, studied as safe realisability, differs from the MST approach of checking conditions during the projection. For an HMSC, it is known that there is a candidate implementation [3], which implements the HMSC if it is implementable. Intuitively, one takes the HMSC and removes all interactions a role is not involved in and determinises the result. We generalise their result to infinite executions.[1]

Hence, algorithms and conditions center around checking implementability of HMSCs. In general, this problem is undecidable [63]. For *globally-cooperative* HMSCs [39], Lohrey [63] proved it to be EXPSPACE-complete. We show that any implementable global type naturally belongs to this class of HMSCs[1] which is far from trivial. These results give rise to the

[1] For this, we impose a mild assumption: all protocols can (but do not need to) terminate.

following algorithm to check implementability of a global type. One can check whether a global type is globally-cooperative (which is equivalent to checking its HMSC encoding). If it is not globally-cooperative, it cannot be implementable. If it is globally-cooperative, we apply the algorithm by Lohrey [63] to check whether its HMSC encoding is implementable. If it is, we use its candidate implementation and know that it generalises to infinite executions.

While this algorithm shows decidability, the complexity might not be tractable. Based on our results, we show how more tractable but still permissive approaches to check implementability of HMSCs can be adapted to the MST setting. In addition, we consider *payload implementability*, which allows to add payload to messages of existing interactions and checks agreement when the additional payload is ignored. We present a sufficient condition for global types that implies payload implementability. These techniques can be used if the previous algorithms are not tractable or reject a global type.

Furthermore, we introduce a generalisation of the implementability problem. A network may reorder messages from different senders for the same receiver but the implementability problem still requires the receiver to receive them in the specified order. Our generalisation allows to consider such reorderings of arrival and can yield performance gains. In addition, it also renders global types implementable that are not implementable in the standard setting. Unfortunately, we prove it to be undecidable in general.

Contributions and Outline

We introduce our MST framework in Section 2 while Section 7 covers details on related work. In the other sections, we introduce the necessary concepts to establish our main *contributions*:

- We give a visual explanation of the classical projection operator with different merge operators and exemplify its shortcomings (Section 3).
- We prove decidability of the implementability problem for global types with generalised choice (Section 4) — provided that protocols can (but do not need to) terminate.
- We comprehensively investigate how MSC techniques can be applied to the MST setting, including algorithmics with better complexity for subclasses as well as an interesting variant of the implementability problem (Section 5).
- Lastly, we introduce a new variant of the implementability problem with a more relaxed role message ordering, which is closer to the network ordering, and prove it to be undecidable in general (Section 6).

2 Multiparty Session Types

In this section, we formally introduce our Multiparty Session Type (MST) framework. We define the syntax of global and local types and their semantics. Subsequently, we recall the implementability problem for global types which asks if there is a deadlock free communicating state machine that admits the same language (without additional synchronisation).

Finite and Infinite Words. Let Σ be an alphabet. We denote the set of finite words over Σ by Σ^* and the set of infinite words by Σ^ω. Their union is denoted by Σ^∞. For two strings $u \in \Sigma^*$ and $v \in \Sigma^\infty$, we say that u is a *prefix* of v if there is some $w \in \Sigma^\infty$ such that $u \cdot w = v$ and denote this with $u \leq v$. For a language $L \subseteq \Sigma^\infty$, we distinguish between the language of finite words $L_{\text{fin}} := L \cap \Sigma^*$ and the language of infinite words $L_{\text{inf}} := L \cap \Sigma^\omega$.

Message Alphabet. We fix a finite set of messages $\mathcal{V}$ and a finite set of roles $\mathcal{P}$, ranged over with $\mathtt{p}$, $\mathtt{q}$, $\mathtt{r}$, and $\mathtt{s}$. With $\Sigma_{sync} = \{\mathtt{p} \to \mathtt{q} : m \mid \mathtt{p}, \mathtt{q} \in \mathcal{P} \text{ and } m \in \mathcal{V}\}$, we denote the set of interactions where sending and receiving a message is specified at the same time. For our asynchronous setting, we also define individual send and receive events: $\Sigma_{\mathtt{p}} = \{\mathtt{p} \triangleright \mathtt{q}!m, \mathtt{p} \triangleleft \mathtt{q}?m \mid \mathtt{q} \in \mathcal{P},\ m \in \mathcal{V}\}$ for a role $\mathtt{p}$. For both send events $\mathtt{p} \triangleright \mathtt{q}!m$ and receive events $\mathtt{p} \triangleleft \mathtt{q}?m$, the first role is *active*, i.e., the sender in the first event and the receiver in the second one. The union for all roles yields all (asynchronous) events: $\Sigma = \bigcup_{\mathtt{p}\in\mathcal{P}} \Sigma_{\mathtt{p}}$. For the rest of this work, we fix the set of roles $\mathcal{P}$, the messages $\mathcal{V}$, and both sets Σ_{sync} and Σ. We may also use the term Σ_{async} for Σ. We define an operator that splits events from Σ_{sync}, $\mathrm{split}(\mathtt{p}\to\mathtt{q}:m) := \mathtt{p} \triangleright \mathtt{q}!m.\, \mathtt{q} \triangleleft \mathtt{p}?m$, which is lifted to sequences and languages as expected. Given a word, we might also project it to all letters of a certain shape. For instance, $w\Downarrow_{\mathtt{p}\triangleright\mathtt{q}!\_}$ is the subsequence of w with all of its send events where $\mathtt{p}$ sends any message to $\mathtt{q}$. If we want to select all messages of w, we write $\mathcal{V}(w)$.

Global and Local Types – Syntax

We give the syntax of global and local types following work by Majumdar et al. [64], Honda et al. [48], Hu and Yoshida [50], as well as Scalas and Yoshida [75]. In this work, we consider global types as specifications for message-passing concurrency and omit features like delegation.

▶ **Definition 2.1** (Syntax of global types). Global types for MSTs *are defined by the grammar:*

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

The term 0 *explicitly represents termination. A term* $\mathtt{p}\to\mathtt{q}_i : m_i$ *indicates an interaction where* $\mathtt{p}$ *sends message* m_i *to* $\mathtt{q}_i$*. In our asynchronous semantics, it is split into a send event* $\mathtt{p}\triangleright\mathtt{q}_i!m_i$ *and a receive event* $\mathtt{q}_i \triangleleft \mathtt{p}?m_i$*. In a choice* $\sum_{i\in I} \mathtt{p}\to\mathtt{q}_i : m_i.G_i$*, the sender* $\mathtt{p}$ *chooses the branch. We require choices to be unique, i.e.,* $\forall i, j \in I.\, i \neq j \Rightarrow \mathtt{q}_i \neq \mathtt{q}_j \vee m_i \neq m_j$*. If* $|I| = 1$*, which means there is no actual choice, we omit the sum operator. The operators* μt *and* t *allow to encode loops. We require them to be guarded, i.e., there must be at least one interaction between the binding* μt *and the use of the recursion variable* t*. Without loss of generality, all occurrences of recursion variables* t *are bound and distinct.*

Our definition allows *generalised choice* as $\mathtt{p}$ can send to different receivers upon branching: $\sum_{i\in I} \mathtt{p}\to\mathtt{q}_i : m_i.G_i$. In contrast, *directed choice* requires a sender to send to a single receiver, i.e., $\forall i, j \in I.\, \mathtt{q}_i = \mathtt{q}_j$.

▶ **Example 2.2** (Global types). The two buyer protocol $\mathbf{G}_{2\mathrm{BP}}$ from the introduction is a global type. Instead of $\sum$, we use $+$ with curly brackets for readability.

▶ **Definition 2.3** (Syntax of local types). *For a role* $\mathtt{p}$*, the* local types *are defined as follows:*

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

We call $\oplus_{i\in I}\, \mathtt{q}_i!m_i$ *an internal choice while* $\&_{i\in I}\, \mathtt{q}_i?m_i$ *is an external choice. For both, we require the choice to be unique, i.e.,* $\forall i, j \in I.\, i \neq j \Rightarrow (\mathtt{q}_i, m_i) \neq (\mathtt{q}_j, m_j)$*. Similarly to global types, we may omit* $\oplus$ *or* $\&$ *if there is no actual choice and we require recursion to be guarded as well as recursion variables to be bound and distinct.*

▶ **Example 2.4** (Local type). For the global type $\mathbf{G}_{2\mathrm{BP}}$, a local type for Seller $\mathtt{s}$ is

$$\mu t.\ \&\begin{cases} \mathtt{a}?query.\,\mathtt{a}!price.\,(\mathtt{a}?buy.\,t\ \&\ \mathtt{a}?no.\,t) \\ \mathtt{a}?done.\,0 \end{cases}.$$

Implementing in a Distributed Setting

Global types can be thought of as global protocol specifications. Thus, a natural question and a main concern in MST theory is whether a global type can be implemented in a distributed setting. We present communicating state machines, which are built from finite state machines, as the standard implementation model.

▶ **Definition 2.5** (State machines [77]). *A* state machine $A = (Q, \Delta, \delta, q_0, F)$ *is a* 5*-tuple with a finite set of states* Q*, an alphabet* Δ*, a transition relation* $\delta \subseteq Q \times (\Delta \cup \{\varepsilon\}) \times Q$*, an initial state* $q_0 \in Q$ *from the set of states, and a set of final states* F *with* $F \subseteq Q$*. If* $(q, a, q') \in \delta$*, we also write* $q \xrightarrow{a} q'$*. A sequence* $q_0 \xrightarrow{w_0} q_1 \xrightarrow{w_1} \ldots$*, with* $q_i \in Q$ *and* $w_i \in \Delta \cup \{\varepsilon\}$ *for* $i \geq 0$*, such that* q_0 *is the initial state, and for each* $i \geq 0$*, it holds that* $(q_i, w_i, q_{i+1}) \in \delta$*, is called a* run *in* A *with its* trace $w_0 w_1 \ldots \in \Delta^\infty$*. A run is* maximal *if it ends in a final state or is infinite. The* language $\mathcal{L}(A)$ *of* A *is the set of traces of all maximal runs. If* Q *is finite, we say* A *is a* finite state machine *(FSM).*

▶ **Definition 2.6** (Communicating state machines [77]). *We call* $\mathcal{A} = \{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p} \in \mathcal{P}}$ *a* communicating state machine *(CSM) over* $\mathcal{P}$ *and* $\mathcal{V}$ *if* $A_\mathtt{p}$ *is a finite state machine with alphabet* $\Sigma_\mathtt{p}$ *for every* $\mathtt{p} \in \mathcal{P}$*. The state machine for* $\mathtt{p}$ *is denoted by* $(Q_\mathtt{p}, \Sigma_\mathtt{p}, \delta_\mathtt{p}, q_{0,\mathtt{p}}, F_\mathtt{p})$*. Intuitively, a CSM allows a set of state machines, one for each role in* $\mathcal{P}$*, to communicate by sending and receiving messages. For this, each pair of roles* $\mathtt{p}, \mathtt{q} \in \mathcal{P}$*,* $\mathtt{p} \neq \mathtt{q}$*, is connected by two directed* message channels*. A transition* $q_\mathtt{p} \xrightarrow{\mathtt{p} \triangleright \mathtt{q}!m} q'_\mathtt{p}$ *in the state machine of* $\mathtt{p}$ *denotes that* $\mathtt{p}$ *sends message* m *to* $\mathtt{q}$ *if* $\mathtt{p}$ *is in the state* $q_\mathtt{p}$ *and changes its local state to* $q'_\mathtt{p}$*. The channel* $\langle \mathtt{p}, \mathtt{q} \rangle$ *is appended by message* m*. For receptions, a transition* $q_\mathtt{q} \xrightarrow{\mathtt{q} \triangleleft \mathtt{p}?m} q'_\mathtt{q}$ *in the state machine of* $\mathtt{q}$ *corresponds to* $\mathtt{q}$ *retrieving the message* m *from the head of the channel when its local state is* $q_\mathtt{q}$ *which is updated to* $q'_\mathtt{q}$*. The run of a CSM always starts with empty channels and each finite state machine is in its respective initial state. A deadlock of* $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p} \in \mathcal{P}}$ *is the last configuration of a finite run for which cannot be extended with* $\rightarrow$*. The formalisation of this intuition is standard and can be found in Appendix A.1.*

A global type always specifies send and receive events together. In a CSM execution, there may be independent events that can occur between a send and its respective receive event.

▶ **Example 2.7** (Motivation for indistinguishability relation $\sim$). Let us consider the following global type which is a part of the two buyer protocol: $\mathtt{a} \to \mathtt{b} : \mathit{cancel}.\, \mathtt{a} \to \mathtt{s} : \mathit{no}.\, 0$. This is one of its traces: $\mathtt{a} \triangleright \mathtt{b}!\mathit{cancel}.\, \mathtt{b} \triangleleft \mathtt{a}?\mathit{cancel}.\, \mathtt{a} \triangleright \mathtt{s}!\mathit{no}.\, \mathtt{s} \triangleleft \mathtt{a}?\mathit{no}$. Because the active roles in $\mathtt{b} \triangleleft \mathtt{a}?\mathit{cancel}$ and $\mathtt{a} \triangleright \mathtt{s}!\mathit{no}$ are different and we do not reorder a receive event in front of its respective send event, any CSM that accepts the previous trace also accepts the following trace: $\mathtt{a} \triangleright \mathtt{b}!\mathit{cancel}.\, \mathtt{a} \triangleright \mathtt{s}!\mathit{no}.\, \mathtt{b} \triangleleft \mathtt{a}?\mathit{cancel}.\, \mathtt{s} \triangleleft \mathtt{a}?\mathit{no}$.

Majumdar et al. [64] introduced the following relation to capture this phenomenon.

▶ **Definition 2.8** (Indistinguishability relation $\sim$ [64]). *We define a family of* indistinguishability relations $\sim_i \subseteq \Sigma^* \times \Sigma^*$*, for* $i \geq 0$ *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* $w.\mathtt{p} \triangleright \mathtt{q}!m.\mathtt{q} \triangleleft \mathtt{p}?m'.u \;\sim_1\; w.\mathtt{q} \triangleleft \mathtt{p}?m'.\mathtt{p} \triangleright \mathtt{q}!m.u$*.*

Let w*,* w'*, and* w'' *be words 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* $v \preceq_\sim u$*. 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 L*, we define* $\mathcal{C}^\sim(L) = \left\{ w' \mid \bigvee \begin{array}{l} w' \in \Sigma^* \wedge \exists w \in \Sigma^*.\ w \in L \text{ and } w' \sim w \\ w' \in \Sigma^\omega \wedge \exists w \in \Sigma^\omega.\ w \in L \text{ and } w' \preceq^\omega_\sim w \end{array} \right\}$*.*

This relation characterises what can be achieved in a distributed setting using CSMs.

▶ **Lemma 2.9** (L. 21 [64]). *Let* $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ *be a CSM. Then,* $\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}) = \mathcal{C}^{\sim}(\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}))$.

Global and Local Types – Semantics

Hence, we define the semantics of global types using the indistinguishability relation $\sim$.

▶ **Definition 2.10** (Semantics of global types). *We construct a state machine* $\mathsf{GAut}(\mathbf{G})$ *to obtain the semantics of a global type* $\mathbf{G}$. *We index every syntactic subterm of* $\mathbf{G}$ *with a unique index to distinguish common syntactic subterms, denoted with* $[G, k]$ *for syntactic subterm* G *and index* k. *Without loss of generality, the index for* $\mathbf{G}$ *is* 0: $[\mathbf{G}, 0]$. *For clarity, we do not quantify indices. We define* $\mathsf{GAut}(\mathbf{G}) = (Q_{\mathsf{GAut}(\mathbf{G})}, \Sigma_{sync}, \delta_{\mathsf{GAut}(\mathbf{G})}, q_{0,\mathsf{GAut}(\mathbf{G})}, F_{\mathsf{GAut}(\mathbf{G})})$ *where*

- $Q_{\mathsf{GAut}(\mathbf{G})}$ *is the set of all indexed syntactic subterms* $[G, k]$ *of* $\mathbf{G}$
- $\delta_{\mathsf{GAut}(\mathbf{G})}$ *is the smallest set containing* $([\sum_{i\in I} \mathtt{p}\to\mathtt{q}_i : m_i.[G_i, k_i], k], \mathtt{p}\to\mathtt{q}_i : m_i, [G_i, k_i])$ *for each* $i \in I$, *and* $([\mu t.[G', k^2], k^1], \varepsilon, [G', k^2])$ *and* $([t, k^3], \varepsilon, [\mu t.[G', k^2], k^1])$,
- $q_{0,\mathsf{GAut}(\mathbf{G})} = [\mathbf{G}, 0]$, *and* $F_{\mathsf{GAut}(\mathbf{G})} = \{[0, k] \mid k$ *is an index for subterm* $0\}$

We consider asynchronous communication so each interaction is split into its send and receive event. In addition, we consider CSMs as implementation model for global types and, from Lemma 2.9, we know that CSM languages are always closed under the indistinguishability relation $\sim$. *Thus, we also apply its closure to obtain the semantics of* $\mathbf{G}$: $\mathcal{L}(\mathbf{G}) := \mathcal{C}^{\sim}(\mathrm{split}(\mathcal{L}(\mathsf{GAut}(\mathbf{G}))$.

The closure $\mathcal{C}^{\sim}$(-) corresponds to similar reordering rules in standard MST developments, e.g., [49, Def. 3.2 and 5.3].

▶ **Example 2.11.** In Figure 1a (p.2), we presented the FSM for the semantics of $\mathsf{GAut}(\mathbf{G}_{2\mathrm{BP}})$. We give the semantics of a simple global type where $\mathtt{p}$ communicates a list of book titles to $\mathtt{q}$: $\mu t.\,(\mathtt{p}\to\mathtt{q}\!:\!\mathit{title}.\,t \;+\; \mathtt{p}\to\mathtt{q}\!:\!\mathit{done}.\,0)$. Its semantics is the union of two cases: if the list of book titles is finite, i.e., $\mathcal{C}^{\sim}((\mathtt{p}\triangleright\mathtt{q}!\mathit{title}.\,\mathtt{q}\triangleleft\mathtt{p}?\mathit{title})^{*}.\,\mathtt{p}\triangleright\mathtt{q}!\mathit{done}.\,\mathtt{q}\triangleleft\mathtt{p}?\mathit{done})$; and the one if the list is infinite, i.e., $\mathcal{C}^{\sim}((\mathtt{p}\triangleright\mathtt{q}!\mathit{title}.\,\mathtt{q}\triangleleft\mathtt{p}?\mathit{title})^{\omega})$. Here, there are only two roles so $\mathcal{C}^{\sim}$(-) can solely delay receive events (Rule 4 of $\sim$).

We distinguish states depending on which subterm they correspond to: *binder states* with their dashed line correspond to a recursion variable binder, while *recursion states* with their dash-dotted lines indicate the use of a recursion variable. We omit ε for transitions from recursion to binder states.

▶ **Definition 2.12** (Semantics for local types). *Given a local type* L *for role* $\mathtt{p}$, *we index syntactic subterms as for the semantics of global types. We construct a state machine* $\mathsf{LAut}(L) = (Q, \Sigma_{\mathtt{p}}, \delta, q_0, F)$ *where*

- Q *is the set of all indexed syntactic subterms in* L,
- δ *is the smallest set containing* $([\oplus_{i\in I}\, \mathtt{q}_i!m_i.[L_i, k_i], k], \mathtt{p}\triangleright\mathtt{q}_i!m_i, [L_i, k_i])$ *and* $([\&_{i\in I}\, \mathtt{q}_i?m_i.[L_i, k_i], k], \mathtt{p}\triangleleft\mathtt{q}_i?m_i, [L_i, k_i])$ *for each* $i \in I$, *as well as* $([\mu t.[L', k^2], k^1], \varepsilon, [L', k^2])$ *and* $([t, k^3], \varepsilon, [\mu t.[L', k^2], k^1])$,
- $q_0 = [L, 0]$ *and* $F = \{[0, k] \mid k$ *is an index for subterm* $0\}$

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

Compared to global types, we distinguish two more kinds of states for local types: a *send state* (internal choice) has a diamond shape while a *receive state* (external choice) has a rectangular shape. For states with ε as next action, we keep the circular shape and call them *neutral states*. Figure 1b (p.2) does not represent the state machine for any local type but illustrates the use of different styles for different kinds of states.

The Implementability Problem for Global Types

The implementability problem for global types asks whether a global type can be implemented in a distributed setting. The projection operator takes the intermediate representation of local types as local specifications for roles. We define implementability directly on the implementation model of CSMs. Intuitively, every set of local types constitutes a CSM through their semantics.

▶ **Definition 2.13** (Implementability [64])**.** *A global type* $\mathbf{G}$ *is said to be* implementable *if there exists a CSM* $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ *such that*

- (deadlock freedom) $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ *is deadlock free, and*
- (protocol fidelity) *their languages are the same:* $\mathcal{L}(\mathbf{G}) = \mathcal{L}(\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}})$*.*

We say that $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ *implements* $\mathbf{G}$*.*

▶ Remark 2.14 (Progress). Deadlock freedom is sometimes also studied as *progress* — in the sense that a system should never get stuck. However, for infinite executions, a role could starve in a non-final state by waiting for a message that is never sent [19, Sec. 3.2]. Castagna et al. [19] consider a stronger notion of progress (Def. 3.3: live session) which requires that every role could eventually reach a final state. Our results apply to this stronger notion of progress as we discuss in Section 4.2.

3 Projection – From Global to Local Types

In this section, we define and visually explain a typical approach to the implementability problem: the classical projection operator. It tries to translate global types to local types and, while doing so, checks if this is safe. Behind the scenes, these checks are conducted by a partial merge operator. We consider different variants of the merge operator from the literature and exemplify the features they support. We provide visual explanations of the classical projection operator with these merge operators on the state machines of global types by example. In Appendix B, we give general descriptions but they are not essential to explain our observations. Lastly, we summarise the shortcomings of the full merge operator and exemplify them with variants of the two buyer protocol from the introduction.

Classical Projection Operator with Parametric Merge

▶ **Definition 3.1** (Projection operator)**.** *For a merge operator* $\sqcap$*, the* projection *of a global type* $\mathbf{G}$ *on to a role* $\mathtt{r}\in\mathcal{P}$ *is a local type that is defined as follows:*[2] $0\!\upharpoonright_\mathtt{r} := 0$ $\qquad t\!\upharpoonright_\mathtt{r} := t$

$$\Big(\textstyle\sum_{i\in I}\mathtt{p}\!\to\!\mathtt{q}\!:\!m_i.G_i\Big)\!\upharpoonright_\mathtt{r} := \begin{cases}\oplus_{i\in I}\,\mathtt{q}!m_i.(G_i\!\upharpoonright_\mathtt{r}) & \textit{if } \mathtt{r}=\mathtt{p}\\ \&_{i\in I}\,\mathtt{p}?m_i.(G_i\!\upharpoonright_\mathtt{r}) & \textit{if } \mathtt{r}=\mathtt{q}\\ \sqcap_{i\in I}\,G_i\!\upharpoonright_\mathtt{r} & \textit{otherwise}\end{cases} \qquad (\mu t.G)\!\upharpoonright_\mathtt{r} := \begin{cases}\mu t.(G\!\upharpoonright_\mathtt{r}) & \textit{if } G\!\upharpoonright_\mathtt{r}\neq t\\ 0 & \textit{otherwise}\end{cases}$$

Intuitively, a projection operator takes the state machine $\mathsf{GAut}(\mathbf{G})$ for a global type $\mathbf{G}$ and projects each transition label to the respective alphabet of the role, e.g., $\mathtt{p}\!\to\!\mathtt{q}\!:\!m$ becomes $\mathtt{q}\triangleleft\mathtt{p}?m$ for role $\mathtt{q}$. This can introduce non-determinism that ought to be resolved by a partial merge operator. Several merge operators have been proposed in the literature.

▶ **Definition 3.2** (Merge Operators)**.** *Let* L_1 *and* L_2 *be local types for a role* $\mathtt{r}$*, and* $\sqcap$ *be a merge operator. We define different cases for the result of* $L_1 \sqcap L_2$*:*

[2] The case split for the recursion binder changes slightly across different definitions. We chose a simple but also the least restrictive condition. We simply check whether the recursion is vacuous (as $\mu t.t$) and omit it in this case. We require to omit μt if t is never used in the result.

(1) L_1 *if* $L_1 = L_2$

$$(2) \begin{pmatrix} \&_{i\in I\setminus J}\, \mathtt{q}?m_i.L'_{1,i} & \& \\ \&_{i\in I\cap J}\, \mathtt{q}?m_i.(L'_{1,i} \sqcap L'_{2,i}) & \& \\ \&_{i\in J\setminus I}\, \mathtt{q}?m_i.L'_{2,i} & \end{pmatrix} \quad \textit{if} \begin{cases} L_1 = \&_{i\in I}\, \mathtt{q}?m_i.L'_{1,i}, \\ L_2 = \&_{i\in J}\, \mathtt{q}?m_i.L'_{2,i} \end{cases}$$

(3) $\mu t_1.(L'_1 \sqcap L'_2[t_2/t_1])$ *if* $L_1 = \mu t_1.L'_1$ *and* $L_2 = \mu t_2.L'_2$

Each merge operator is defined by a collection of cases it can apply. If none of the respective cases applies, the result of the merge is undefined. The plain merge $\sqcap_{\mathsf{p}}$ [28] *can only apply Case* (1)*. The semi-full merge* $\sqcap_{\mathsf{s}}$ [85] *can apply Cases* (1) *and* (2)*. The full merge* $\sqcap_{\mathsf{f}}$ [75] *can apply all Cases* (1)*,* (2)*, and* (3)*.*

We will also consider the availability merge operator $\sqcap_{\mathsf{a}}$ by Majumdar et al. [64] which builds on the full merge operator but generalises Case (2) to allow generalised choice. We will explain the main differences in Remark 3.12.

▶ Remark 3.3 (Correctness of projection). This would be the correctness criterion for projection: Let $\mathbf{G}$ be some global type and let plain merge $\sqcap_{\mathsf{p}}$, semi full merge $\sqcap_{\mathsf{s}}$, full merge $\sqcap_{\mathsf{f}}$, or availability merge $\sqcap_{\mathsf{a}}$ be the merge operator $\sqcap$. If $\mathbf{G}{\restriction_{\mathtt{p}}}$ is defined for each role $\mathtt{p}$, then the CSM $\{\!\!\{\mathsf{LAut}(\mathbf{G}{\restriction_{\mathtt{p}}})\}\!\!\}_{\mathtt{p}\in\mathcal{P}}$ implements $\mathbf{G}$.
We do not actually prove this so we do not state it as lemma. *But why does this hold?* The implementability condition is the combination of deadlock freedom and protocol fidelity. Coppo et al. [28] show that *subject reduction* entails protocol fidelity and progress while progress, in turn, entails deadlock freedom. Subject reduction has been proven for the plain merge operator [28, Thm. 1] and the semi-full operator [85, Thm. 1]. Scalas and Yoshida pointed out that several versions of classical projection with the full merge are flawed [75, Sec. 8.1]. Hence, we have chosen a full merge operator whose correctness follows from the correctness of the more general availability merge operator. For the latter, the correctness follows from work by Majumdar et al. [64, Thm. 16].

▶ **Example 3.4** (Projection without merge / Collapsing erasure)**.** In the introduction, we considered $\mathbf{G}_{2\mathrm{BP}}$ and the FSM for its semantics in Figure 1a. We projected (without merge) on to Seller $\mathtt{s}$ to obtain the FSM in Figure 1b. In general, we also collapse neutral states with a single ε-transition and their only successor. We call this *collapsing erasure*. We only need to actually collapse states for the protocol in Figure 4a. In all other illustrations, we indicate the interactions the role is not involved with the following notation: $[\mathtt{p}\!\to\!\mathtt{q}\!:\!l] \rightsquigarrow \varepsilon$.

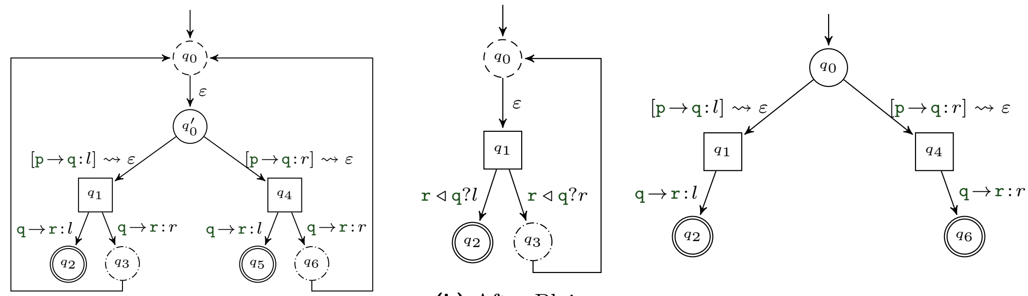
On the Structure of $\mathsf{GAut}(\mathbf{G})$

We now show that the state machine for every local and global type has a certain shape. This simplifies the visual explanations of the different merge operators. Intuitively, every such state machine has a tree-like structure where backward transitions only happen at leaves of the tree, are always labelled with ε, and only lead to ancestors. The FSM in Figure 1a (p.2) illustrates this shape where the root of the tree is at the top.

▶ **Definition 3.5** (Ancestor-recursive, non-merging, intermediate recursion, etc.)**.** *Let* $A = (Q, \Delta, \delta, q_0, F)$ *be a finite state machine. We say that* A *is* ancestor-recursive *if there is a function* $\mathrm{lvl}\colon Q \to \mathbb{N}$ *such that, for every transition* $q \xrightarrow{x} q' \in \delta$*, one of the two holds:*

(1) $\mathrm{lvl}(q) > \mathrm{lvl}(q')$*, or*

(2) $x = \varepsilon$ *and there is a run from the initial state* q_0 *(without going through* q*) to* q' *which can be completed to reach* q*:* $q_0 \xrightarrow{-} \ldots \xrightarrow{-} q_n$ *is a run with* $q_n = q'$ *and* $q \neq q_i$ *for every* $0 \le i \le n$*, and the run can be extended to* $q_0 \xrightarrow{-} \ldots \xrightarrow{-} q_n \xrightarrow{-} \ldots \xrightarrow{-} q_{n+m}$ *with* $q_{n+m} = q$*. Then, the state* q' *is called* ancestor *of* q*.*



(a) Positive Example for Plain Merge **(b)** After Plain Merge **(c)** Negative example for plain merge

Figure 2 The FSM on the left represents an implementable global type that is accepted by plain merge. It implicitly shows the FSM after collapsing erasure: every interaction $\mathtt{r}$ is not involved in is given as $[\mathtt{p}\to\mathtt{q}:l] \rightsquigarrow \varepsilon$. The FSM in the middle is the result of the plain merge. The FSM on the right represents an implementable global type that is rejected by plain merge. It is obtained from the left one by removing one choice option in each branch of the initial choice.

We call the first (1) *kind of transition* forward transition *while the second* (2) *kind is a* backward transition*. The state machine A is said to be free from* intermediate recursion *if every state q with more than one outgoing transition, i.e., $|\{q' \mid q \xrightarrow{-} q' \in \delta\}| > 1$, has only forward transitions. We say that A is* non-merging *if every state only has one incoming edge with greater level, i.e., for every state q', $\{q \mid q \xrightarrow{-} q' \in \delta \wedge \mathrm{lvl}(q) > \mathrm{lvl}(q')\} \leq 1$. The state machine A is* dense *if, for every $q \xrightarrow{x} q' \in \delta$, the transition label x is ε implies that q has only one outgoing transition. Last, the* cone *of q are all states q' which are reachable from q and have a smaller level than q, i.e.,* $\mathrm{lvl}(q) > \mathrm{lvl}(q')$.

▶ **Proposition 3.6** (Shape of $\mathsf{GAut}(\mathbf{G})$ and $\mathsf{LAut}(L)$)**.** *Let $\mathbf{G}$ be some global type and L be some local type. Then, both $\mathsf{GAut}(\mathbf{G})$ and $\mathsf{LAut}(L)$ are ancestor-recursive, free from intermediate recursion, non-merging, and dense.*

For both, the only forward ε-transitions occur precisely from binder states while backward transitions happen from variable states to binder states. The illustrations for our examples always have the initial state, which is the state with the greatest level, at the top. This is why we use greater and higher as well as smaller and lower interchangeably for levels.

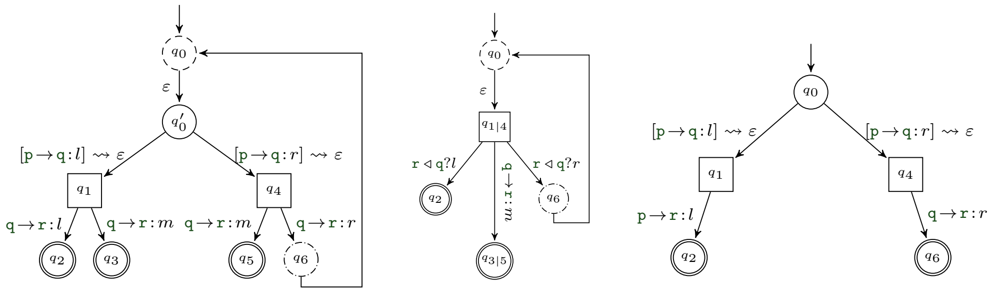
Features of Different Merge Operators by Example

In this section, we exemplify which features each of the merge operators does support. We present a sequence of implementable global types. Despite, some cannot be handled by some (or all) merge operators. If a global type is not projectable using some merge operator, we say it is *rejected* and a *negative* example for this merge operator. We focus on role $\mathtt{r}$ when projecting. Thus, rejected mostly means that there is (at least) no projection on to $\mathtt{r}$. If a global type is projectable by some merge operator, we call it a *positive* example. All examples strive for minimality and follow the idea that roles decide whether to take a left (l) or right (r) branch of a choice.

▶ **Example 3.7** (Positive example for plain merge)**.** The following global type is implementable:

$$\mu t. + \begin{cases} \mathtt{p}\to\mathtt{q}:l.\,(\mathtt{q}\to\mathtt{r}:l.\,0 + \mathtt{q}\to\mathtt{r}:r.\,t) \\ \mathtt{p}\to\mathtt{q}:r.\,(\mathtt{q}\to\mathtt{r}:l.\,0 + \mathtt{q}\to\mathtt{r}:r.\,t) \end{cases} .$$

The state machine for its semantics is given in Figure 2a. After collapsing erasure, there is a non-deterministic choice from q'_0 leading to q_1 and q_4 since $\mathtt{r}$ is not involved in the initial choice. The plain merge operator can resolve this non-determinism since both cones of q_1



(a) Positive example for semi-full merge **(b)** After semi-full merge **(c)** Negative example for full merge

Figure 3 The FSM on the left represents an implementable global type (and implicitly the collapsing erasure on to $\mathtt{r}$) that is accepted by semi-full merge. The FSM in the middle is the result of the semi-full merge. The FSM on the right is a negative example for the full merge operator.

and q_4 represent the same subterm. Technically, there is an isomorphism between the states in both cones which preserves the kind of states as well as the transition labels and the backward transitions from isomorphic recursion states lead to the same binder state. The result is illustrated in Figure 2b. It is also the FSM of a local type for $\mathtt{r}$ which is the result of the (syntactic) plain merge: $\mu t.(\mathtt{r} \triangleleft \mathtt{q}?l.\, 0 \,\&\, \mathtt{r} \triangleleft \mathtt{q}?r.\, t)$.

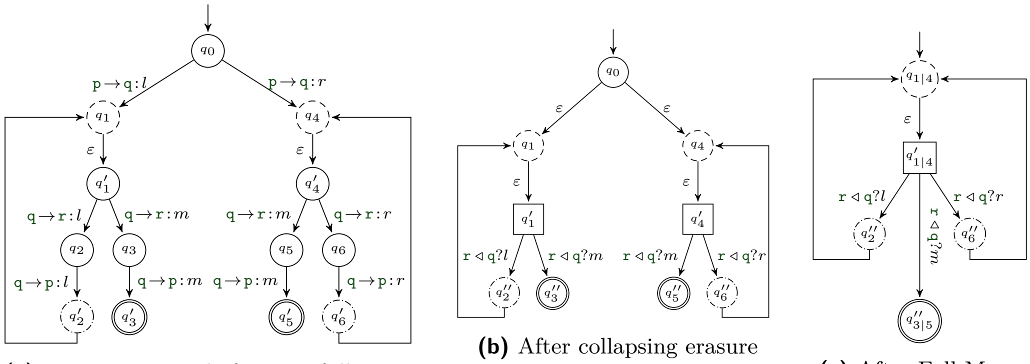
Our explanation on FSMs allows to check congruence of cones to merge while the definition requires syntactic equality. If we swap the order of branches $\mathtt{q}\to\mathtt{r}:l$ and $\mathtt{q}\to\mathtt{r}:r$ in Figure 2a on the right, the syntactic merge rejects. Still, because both are semantically the same protocol specification, we expect tools to check for such easy fixes.

▶ **Example 3.8** (Negative example for plain merge)**.** We consider the following simple implementable global type where the choice by $\mathtt{p}$ is propagated to $\mathtt{r}$: $+\begin{cases}\mathtt{p}\to\mathtt{q}:l.\,\mathtt{q}\to\mathtt{r}:l.\,0\\ \mathtt{p}\to\mathtt{q}:r.\,\mathtt{q}\to\mathtt{r}:r.\,0\end{cases}$. The corresponding state machine is illustrated in Figure 2c. Here, q_0 exhibits non-determinism but the plain merge fails because q_1 and q_4 have different outgoing transition labels.

Intuitively, the plain merge operator forbids that any, but the two roles involved in a choice, can have different behaviour after the choice. It basically forbids propagating a choice. The semi-full merge overcomes this shortcoming and can handle the previous example. We present a slightly more complex one to showcase the features it supports.

▶ **Example 3.9** (Positive example for semi-full merge)**.** Let us consider the following implementable global type: $\mu t.+\begin{cases}\mathtt{p}\to\mathtt{q}:l.\,(\mathtt{q}\to\mathtt{r}:l.\,0+\mathtt{q}\to\mathtt{r}:m.\,0)\\ \mathtt{p}\to\mathtt{q}:r.\,(\mathtt{q}\to\mathtt{r}:m.\,0+\mathtt{q}\to\mathtt{r}:r.\,t)\end{cases}$. The state machine for its semantics is illustrated in Figure 3a. After applying collapsing erasure, there is a non-deterministic choice from q_0 leading to q_1 and q_4 since $\mathtt{r}$ is not involved in the initial choice, We apply the semi-full merge for both states. Both are receive states so Case (2) applies. First, we observe that $\mathtt{r} \triangleleft \mathtt{q}?l$ and $\mathtt{r} \triangleleft \mathtt{q}?r$ are unique to one of the two states so both transitions, with the cones of the states they lead to, can be kept. Second, there is $\mathtt{r} \triangleleft \mathtt{q}?m$ which is possible in both states. We recursively apply the semi-full merge and, with Case (1), observe that the result $q_{3,5}$ is simply a final state. Overall, we obtain the state machine in Figure 3b which is equivalent to the result of the syntactic projection with semi-full merge: $\mu t.(\mathtt{r} \triangleleft \mathtt{q}?l.\,0 + \mathtt{r} \triangleleft \mathtt{q}?m.\,0 + \mathtt{r} \triangleleft \mathtt{q}?r.\,t)$.

▶ **Example 3.10** (Negative example for semi-full merge and positive example for full merge)**.** The semi-full merge operator rejects the following implementable global type:



(a) Negative example for semi-full merge and positive example for full merge

(b) After collapsing erasure

(c) After Full Merge

Figure 4 The FSM on the left represents an implementable global type that is rejected by the semi-full merge. It is accepted by the full merge: collapsing erasure yields the FSM in the middle and applying the full merge the FSM on the right.

$$+\begin{cases} \mathtt{p}\to\mathtt{q}:l.\ \mu t_1.\,(\mathtt{q}\to\mathtt{r}:l.\ \mathtt{q}\to\mathtt{p}:l.\ t_1 + \mathtt{q}\to\mathtt{r}:m.\ \mathtt{q}\to\mathtt{p}:m.\ 0)\\ \mathtt{p}\to\mathtt{q}:r.\ \mu t_2.\,(\mathtt{q}\to\mathtt{r}:m.\ \mathtt{q}\to\mathtt{p}:m.\ 0 + \mathtt{q}\to\mathtt{r}:r.\ \mathtt{q}\to\mathtt{p}:r.\ t_2)\end{cases}.$$

Its FSM and the FSM after collapsing erasure is given in Figures 4a and 4b. Intuitively, it would need to recursively merge the parts after both recursion binders in order to merge the branches with receive event $\mathtt{r} \triangleleft \mathtt{q}?m$ but it cannot do so. The full merge can handle this global type. It can descend beyond q_1 and q_4 and is able to merge q'_1 and q'_4. To obtain $q''_{3|5}$, it applies Case (1) while $q'_{1|4}$ is only feasible with Case (2). The result is embedded into the recursive structure to obtain the FSM in Figure 4c. It is equivalent to the (syntactic) result which renames the recursion variable for one branch: $\mu t_1.\,(\mathtt{r} \triangleleft \mathtt{q}?l.\ t_1 \,\&\, \mathtt{r} \triangleleft \mathtt{q}?m.\ 0 \,\&\, \mathtt{r} \triangleleft \mathtt{q}?r.\ t_1)$.

▶ **Example 3.11** (Negative example for full merge)**.** We consider a simple implementable global type where $\mathtt{p}$ propagates its decision to $\mathtt{r}$ in the top branch while $\mathtt{q}$ propagates it in the bottom branch: $+\begin{cases}\mathtt{p}\to\mathtt{q}:l.\ \mathtt{p}\to\mathtt{r}:l.\ 0\\ \mathtt{p}\to\mathtt{q}:r.\ \mathtt{q}\to\mathtt{r}:r.\ 0\end{cases}$. It is illustrated in Figure 3c. This cannot be projected on to $\mathtt{r}$ by the full merge operator for which all receive events need to have the same sender.

▶ Remark 3.12 (On generalised choice). Majumdar et al. [64] proposed a classical projection operator that allows to overcome this shortcoming. It can project the previous example. In general, allowing to receive from different senders has subtle consequences. Intuitively, messages from different senders could overtake each other in a distributed setting and one cannot rely on the FIFO order provided by the channel of a single sender. Thus, they employ a message availability analysis to ensure that there cannot be any confusion about which branch shall be taken. Except for the possibility to merge cases where a receiver receives from multiple senders, their merge operator suffers from the same shortcomings as any classical projection operator. We refrain from presenting their merge operator here but refer to their work for details on the availability merge operator ◪.

Case (2) allows to descend for common receive events. One could also add a similar case for send events where one recursively applies the merge operator (but, in most cases, the set of send events ought to be the same). Such a case might render some global types projectable. However, it does not give any additional insights into the concept of the classical projection operator and its potential merge operators. Of course, one could consider the different cases in all combinations. Again, this does not really give insights which is why we deliberately chose this incremental style that concisely shows which cases support which features.

Shortcomings of Classical Projection/Merge Operators

In this section, we present slight variations of the two buyer protocol that are implementable but rejected by all of the presented projection/merge operators.

▶ **Example 3.13.** We obtain an implementable variant by omitting both message interactions $\mathtt{a} \to \mathtt{s} : no$ with which Buyer a notifies Seller s that they will not buy the item:

$$\mu t.\ + \begin{cases} \mathtt{a} \to \mathtt{s} : query.\ \mathtt{s} \to \mathtt{a} : price.\ \big(\mathtt{a} \to \mathtt{b} : split.\ (\mathtt{b} \to \mathtt{a} : yes.\ \mathtt{a} \to \mathtt{s} : buy.\ t + \mathtt{b} \to \mathtt{a} : no.\ t) + \mathtt{a} \to \mathtt{b} : cancel.\ t\big) \\ \mathtt{a} \to \mathtt{s} : done.\ \mathtt{a} \to \mathtt{b} : done.\ 0 \end{cases} .$$

This global type cannot be projected on to Seller s. The merge operator would need to merge a recursion variable with an external choice. Visually, the merge operator does not allow to unfold the variable t and try to merge again. However, there is a local type:

$$\mu t_1.\ \& \begin{cases} \mathtt{s} \triangleleft \mathtt{a}?query.\ \mu t_2.\ \mathtt{s} \triangleright \mathtt{a}!price.\ (\mathtt{s} \triangleleft \mathtt{a}?buy.\ t_1 \ \&\ \mathtt{s} \triangleleft \mathtt{a}?query.\ t_2 \ \&\ \mathtt{s} \triangleleft \mathtt{a}?done.\ 0) \\ \mathtt{s} \triangleleft \mathtt{a}?done.\ 0 \end{cases} .$$

The local type has two recursion variable binders while the global type only has one. Our explanations showed that classical projection operators can never yield such a structural change: the merge operator can only merge states but not introduce new ones or introduce new backward transitions.

▶ **Example 3.14** (Two Buyer Protocol with Subscription)**.** In this variant, Buyer a first decides whether to subscribe to a yearly discount offer or not — before purchasing the sequence of items — and notifies Buyer b if it does so: $\mathbf{G}_{2\mathrm{BPWS}} \ := \ + \begin{cases} \mathtt{a} \to \mathtt{s} : login.\ \mathbf{G}_{2\mathrm{BP}} \\ \mathtt{a} \to \mathtt{s} : subscribe.\ \mathtt{a} \to \mathtt{b} : subscribed.\ \mathbf{G}_{2\mathrm{BP}} \end{cases}.$ The merge operator needs to merge a recursion variable binder μt with an external choice $\mathtt{b} \triangleleft \mathtt{a}?subscribed$. Because Buyer a only sends *subscribed* at the beginning of the protocol, it is safe to introduce one recursion variable earlier to obtain the following local type for Buyer b. (In fact, we could also remove μt_2 and substitute t_2 by t_1 for the same reason.)

$$\mu t_1.\ \& \begin{cases} \mathtt{b} \triangleleft \mathtt{a}?split.\ (\mathtt{b} \triangleright \mathtt{a}!yes.\ t_1 \oplus \mathtt{b} \triangleright \mathtt{a}!no.\ t_1) \\ \mathtt{b} \triangleleft \mathtt{a}?cancel.\ t_1 \\ \mathtt{b} \triangleleft \mathtt{a}?done.\ 0 \\ \mathtt{b} \triangleleft \mathtt{a}?subscribed.\ \mu t_2.\ \big(\mathtt{b} \triangleleft \mathtt{a}?split.\ (\mathtt{b} \triangleright \mathtt{a}!yes.\ t_2 \oplus \mathtt{b} \triangleright \mathtt{a}!no.\ t_2) \ \&\ \mathtt{b} \triangleleft \mathtt{a}?cancel.\ t_2 \ \&\ \mathtt{b} \triangleleft \mathtt{a}?done.\ 0\big) \end{cases} .$$

Similarly, the classical projection operator cannot yield any local type which needs to distinguish semantic properties to disambiguate a choice, e.g., counting modulo a constant. Scalas and Yoshida [75] identified another shortcoming: most classical projection operators require all branches of a loop to contain the same set of active roles. Thus, they cannot project the following global type. It is implementable and if it was projectable, the result would be equivalent to the local types given in their example [75, Fig. 4 (2)].

▶ **Example 3.15** (Two Buyer Protocol with Inner Recursion)**.** This variant allows to recursively negotiate how to split the price (and omits the outer recursion):

$$\mathbf{G}_{2\mathrm{BPIR}} \quad := \quad \mathtt{a} \to \mathtt{s} : query.\ \mathtt{s} \to \mathtt{a} : price.\ \mu t.\ + \begin{cases} \mathtt{a} \to \mathtt{b} : split.\ (\mathtt{b} \to \mathtt{a} : yes.\ \mathtt{a} \to \mathtt{s} : buy.\ 0 + \mathtt{b} \to \mathtt{a} : no.\ t) \\ \mathtt{a} \to \mathtt{b} : cancel.\ \mathtt{a} \to \mathtt{s} : no.\ 0 \end{cases} .$$

These shortcomings have been addressed by some non-classical approaches. For example, Scalas and Yoshida [75] employ model checking while Dagnino et al. [31] characterise implementable global types with an undecidable well-formedness condition and give a sound algorithmically checkable approximation. It is not known whether the implementability problem for global types, neither with directed or generalised choice, is decidable. We answer this question positively for the more general case of generalised choice.

4 Implementability for Global Types from MSTs is Decidable

In this section, we show that the implementability problem for global types with generalised choice is decidable. For this, we use results from the domain of message sequence charts.

We first introduce high-level message sequence charts (HMSCs) and recall an encoding of global types to HMSCs. In general, implementability of HMSCs is undecidable but we prove that global types belong, when encoded as HMSCs, to a class of HMSCs for which implementability is decidable.

4.1 High-level Message Sequence Charts

Our definitions of (high-level) message sequence charts follow work by Genest et al. [38] and Stutz and Zufferey [77]. If reasonable, we adapt terminology to the MST setting.

▶ **Definition 4.1** (Message Sequence Charts). *A* message sequence chart (MSC) *is a* 5*-tuple* $M = (N, p, f, l, (\leq_{\mathtt{p}})_{\mathtt{p}\in\mathcal{P}})$ *where*

- N *is a set of send* (S) *and receive* (R) *event nodes such that* $N = S \uplus R$ *(where* $\uplus$ *denotes disjoint union),*
- $p\colon N \to \mathcal{P}$ *maps each event node to the role acting on it,*
- $f\colon S \to R$ *is an injective function linking corresponding send and receive event nodes,*
- $l\colon N \to \Sigma$ *labels every event node with an event, and*
- $(\leq_{\mathtt{p}})_{\mathtt{p}\in\mathcal{P}}$ *is a family of total orders for the event nodes of each role:* $\leq_{\mathtt{p}} \subseteq p^{-1}(\mathtt{p}) \times p^{-1}(\mathtt{p})$.

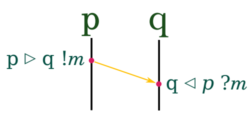


Figure 5 Highlighting the elements of a MSC: $(N, p, f, l, (\leq_{\mathtt{p}})_{\mathtt{p}\in\mathcal{P}})$

An MSC M *induces a partial order* $\leq_M$ *on* N *that is defined co-inductively*:

$$\frac{e \leq_{\mathtt{p}} e'}{e \leq_M e'}\ \textit{PROC} \qquad \frac{s \in S}{s \leq_M f(s)}\ \textit{SND-RCV} \qquad \frac{}{e \leq_M e}\ \textit{REFL} \qquad \frac{e \leq_M e' \quad e' \leq_M e''}{e \leq_M e''}\ \textit{TRANS}$$

The labelling function l *respects the function* f*: for every send event node* e*, we have that* $l(e) = p(e) \triangleright p(f(e))!m$ *and* $l(f(e)) = p(f(e)) \triangleleft p(e)?m$ *for some* $m \in \mathcal{V}$.

All MSCs in our work respect FIFO, i.e., there are no $\mathtt{p}$ and $\mathtt{q}$ such that there are $e_1, e_2 \in p^{-1}(\mathtt{p})$ with $e_1 \neq e_2$, $l(e_1) = l(e_2)$, $e_1 \leq_{\mathtt{p}} e_2$ and $f(e_2) \leq_{\mathtt{q}} f(e_1)$ (also called degenerate) and for every pair of roles $\mathtt{p}$, $\mathtt{q}$, and for every two event nodes $e_1 \leq_M e_2$ with $l(e_i) = \mathtt{p} \triangleright \mathtt{q}!\_$ for $i \in \{1, 2\}$, it holds that $\mathcal{V}(w_{\mathtt{p}}) = \mathcal{V}(f(w_{\mathtt{p}}))$ where $w_{\mathtt{p}}$ is the (unique) linearisation of $p^{-1}(\mathtt{p})$. A *basic MSC (BMSC)* has a finite number of nodes N and $\mathcal{M}$ denotes the set of all BMSCs. When unambiguous, we omit the index M for $\leq_M$ and write $\leq$. We define $\lneq$ as expected. The language $\mathcal{L}(M)$ of an MSC M collects all words $l(w)$ for which w is a linearisation of N that is compliant with $\leq_M$.

If one thinks of a BMSC as straight-line code, a high-level message sequence chart adds control flow. It embeds BMSCs into a graph structure which allows for choice and recursion.

▶ **Definition 4.2** (High-level Message Sequence Charts [77]). *A* high-level message sequence chart (HMSC) *is a* 5*-tuple* (V, E, v^I, V^T, μ) *where* V *is a finite set of vertices,* $E \subseteq V \times V$ *is a set of directed edges,* $v^I \in V$ *is an initial vertex,* $V^T \subseteq V$ *is a set of terminal vertices, and* $\mu : V \to \mathcal{M}$ *is a function mapping every vertex to a BMSC. A path in an HMSC is a sequence of vertices* $v_1, \ldots$ *from* V *that is connected by edges, i.e.,* $(v_i, v_{i+1}) \in E$ *for every* i*. A path is* maximal *if it is infinite or ends in a vertex from* V^T.

Intuitively, the language of an HMSC is the union of all languages of the finite and infinite MSCs generated from maximal paths in the HMSC and is formally defined in Appendix C.1.

Like global types, an HMSC specifies a protocol. The implementability question was also posed for HMSCs and studied as *safe realisability.* If the CSM is not required to be deadlock free, it is called weak realisability.

▶ **Definition 4.3** (Safe realisability of HMSCs [4]). *An HMSC* H *is said to be* safely realisable *if there exists a deadlock free CSM* $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ *such that* $\mathcal{L}(H) = \mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}})$.

Encoding Global Types from MSTs as HMSCs

Stutz and Zufferey [77] provide a formal connection from global types to HMSCs. We recall their encoding and main correctness result.

▶ **Definition 4.4** (Encoding global types as HMSCs [77]). *In the translation, the following notation is used: $M_\emptyset$ is the empty BMSC ($N = \emptyset$) and $M(\mathtt{p}\to\mathtt{q}:m)$ is the BMSC with two event nodes: e_1, e_2 such that $f(e_1) = e_2$, $l(e_1) = \mathtt{p} \triangleright \mathtt{q}!m$, and $l(e_2) = \mathtt{q} \triangleleft \mathtt{p}?m$.*

Let $\mathbf{G}$ be a global type, we construct an HMSC $H(\mathbf{G}) = (V, E, v^I, V^T, \mu)$ with

$$
\begin{aligned}
V = \;& \{G' \mid G' \text{ is a subterm of } \mathbf{G}\} \cup \\
& \{(\textstyle\sum_{i\in I} \mathtt{p}\to\mathtt{q}_i : m_i.G_i, j) \mid \textstyle\sum_{i\in I} \mathtt{p}\to\mathtt{q}_i : m_i.G_i \text{ occurs in } \mathbf{G} \wedge j \in I\} \\
E = \;& \{(\mu t.G', G') \mid \mu t.G' \text{ occurs in } \mathbf{G}\} \cup \{(t, \mu t.G') \mid t, \mu t.G' \text{ occurs in } \mathbf{G}\} \\
& \cup \{(\textstyle\sum_{i\in I} \mathtt{p}\to\mathtt{q}_i : m_i.G_i, (\textstyle\sum_{i\in I} \mathtt{p}\to\mathtt{q}_i : m_i.G_i, j)) \mid (\textstyle\sum_{i\in I} \mathtt{p}\to\mathtt{q}_i : m_i.G_i, j) \in V\} \\
& \cup \{((\textstyle\sum_{i\in I} \mathtt{p}\to\mathtt{q}_i : m_i.G_i, j), G_j) \mid (\textstyle\sum_{i\in I} \mathtt{p}\to\mathtt{q}_i : m_i.G_i, j) \in V\} \\
v^I = \;& \mathbf{G} \qquad V^T = \{0\} \qquad \mu(v) = \begin{cases} M(\mathtt{p}\to\mathtt{q}_i : m_j) & \text{if } v = (\sum_{i\in I} \mathtt{p}\to\mathtt{q}_i : m_i.G_i\}, j) \\ M_\emptyset & \text{otherwise} \end{cases}
\end{aligned}
$$

We adapt the correctness result to our definitions. In particular, our semantics of $\mathbf{G}$ use the closure operator $\mathcal{C}^{\sim}(\text{-})$ while they explicitly distinguish between a type and execution language. We also omit the closure operator on the right-hand side because HMSCs are closed with regard to this operator [77, Lm. 5].

▶ **Theorem 4.5.** *Let $\mathbf{G}$ be a global type. Then, the following holds: $\mathcal{L}(\mathbf{G}) = \mathcal{L}(H(\mathbf{G}))$.*

4.2 Implementability is Decidable

▶ **Assumption** (0-Reachable). *We introduce a mild assumption for global types. We say a global type $\mathbf{G}$ is 0-reachable if every prefix of a word in its language can be completed to a finite word. Equivalently, we require that the vertex 0 is reachable from any vertex in $H(\mathbf{G})$.*[3] *Intuitively, this solely rules out global types that have loops without exit (cf. Example 4.19).*

The MSC approach to safe realisability for HMSCs is different from the classical projection approach to implementability. Given an HMSC, there is a canonical candidate implementation which always implements the HMSC if an implementation exists [3, Thm. 13]. Therefore, approaches center around checking safe realisability of HMSC languages and establishing conditions on HMSCs that entail safe realisability.

▶ **Definition 4.6** (Canonical candidate implementation [3]). *Given an HMSC H and a role $\mathtt{p}$, let $A'_\mathtt{p} = (Q', \Sigma_\mathtt{p}, \delta', q'_0, F')$ be a state machine with $Q' = \{q_w \mid w \in \operatorname{pref}(\mathcal{L}(H){\Downarrow_{\Sigma_\mathtt{p}}})\}$, $F' = \{q_w \mid w \in \mathcal{L}_{\mathrm{fin}}(H){\Downarrow_{\Sigma_\mathtt{p}}})\}$, and $\delta'(q_w, x, q_{wx})$ for $x \in \Sigma_{async}$. The resulting state machine $A'_\mathtt{p}$ is not necessarily finite so $A'_\mathtt{p}$ is determinised and minimised which yields the FSM $A_\mathtt{p}$. We call $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ the canonical candidate implementation of H.*

Intuitively, the intermediate state machine $A'_\mathtt{p}$ constitutes a tree whose maximal finite paths give $\mathcal{L}(H){\Downarrow_{\Sigma_\mathtt{p}}} \cap \Sigma_\mathtt{p}^*$. This set can be infinite and, thus, the construction might not be effective. We give an effective construction of a deterministic FSM for the same language which was very briefly hinted at by Alur et al. [4, Proof of Thm. 3].

[3] An equivalent conditions is common in the HMSC domain [39, Sec. 2] [77, Sec. 4].

▶ **Definition 4.7** (Projection by Erasure). *Let* $\mathtt{p}$ *be a role and* $M = (N, p, f, l, (\leq_{\mathtt{p}})_{\mathtt{p}\in\mathcal{P}})$ *be an MSC. We denote the set of nodes of* $\mathtt{p}$ *with* $N_{\mathtt{p}} := \{n \mid p(n) = \mathtt{p}\}$ *and define a two-ary* next*-relation on* $N_{\mathtt{p}}$: $\operatorname{next}(n_1, n_2)$ *iff* $n_1 \lneq n_2$ *and there is no* n' *with* $n_1 \lneq n' \lneq n_2$*. We define the projection by erasure of* M *on to* $\mathtt{p}$*:* $M{\Downarrow_{\mathtt{p}}} = (Q_M, \Sigma_{\mathtt{p}}, \delta_M, q_{M,0}, \{q_{M,f}\})$ *with*

$$Q_M = \{q_n \mid n \in N_{\mathtt{p}}\} \uplus \{q_{M,0}\} \uplus \{q_{M,f}\} \text{ and}$$

$$\delta_M = \{q_{n_1} \xrightarrow{l(n_1)} q_{n_2} \mid \operatorname{next}(n_1, n_2)\} \uplus \{q_{n_2} \xrightarrow{l(n_2)} q_{M,f} \mid \forall n_1.\, n_1 \leq n_2\} \uplus \{q_{M,0} \xrightarrow{\varepsilon} q_{n_1} \mid \forall n_2.\, n_1 \leq n_2\}$$

where $\uplus$ *denotes disjoint union. Let* $H = (V, E, v^I, V^T, \mu)$ *be an HMSC. We construct the projection by erasure for every vertex and identify them with the vertex, e.g.,* Q_v *instead of* $Q_{\mu(v)}$*. We construct an auxiliary FSM* $(Q'_H, \Sigma_{\mathtt{p}}, \delta'_H, q'_{H,0}, F'_H)$ *with* $Q'_H = \biguplus_{v\in V} Q_v$*,* $\delta'_H = \biguplus_{v\in V} \delta_v \uplus \{q_{v_1,f} \xrightarrow{\varepsilon} q_{v_2,0} \mid (v_1, v_2) \in E\}$*,* $q'_{H,0} = q_{v^I,0}$*, and* $F'_H = \biguplus_{v\in V^F} q_{v,f}$*. We determinise and minimise* $(Q'_H, \Sigma_{\mathtt{p}}, \delta'_H, q'_{H,0}, F'_H)$ *to obtain* $H{\Downarrow_{\mathtt{p}}} := (Q_H, \Sigma_{\mathtt{p}}, \delta_H, q_{H,0}, F_H)$ *which we define to be the* projection by erasure of H on to $\mathtt{p}$*. The CSM formed from the projections by erasure* $\{\!\{H{\Downarrow_{\mathtt{p}}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ *is called* erasure candidate implementation.

▶ **Lemma 4.8** (Correctness of Projection by Erasure). *Let* H *be an HMSC,* $\mathtt{p}$ *be a role, and* $H{\Downarrow_{\mathtt{p}}}$ *be its projection by erasure. Then, the following language equality holds:* $\mathcal{L}(H{\Downarrow_{\mathtt{p}}}) = \mathcal{L}(H){\Downarrow_{\Sigma_{\mathtt{p}}}}$.

The proof is straightforward and can be found in Appendix C.2. From this result and the construction of the canonical candidate implementation, it follows that the projection by erasure admits the same finite language.

▶ **Corollary 4.9.** *Let* H *be an HMSC,* $\mathtt{p}$ *be a role,* $H{\Downarrow_{\mathtt{p}}}$ *be its projection by erasure, and* $A_{\mathtt{p}}$ *be the canonical candidate implementation. Then, it holds that* $\mathcal{L}_{\text{fin}}(H{\Downarrow_{\mathtt{p}}}) = \mathcal{L}_{\text{fin}}(A_{\mathtt{p}})$.

The projection by erasure can be computed effectively and is also deterministic. Thus, we use it in place of the canonical candidate implementation. Given a global type, the erasure candidate implementation for its HMSC encoding implements it if it is implementable.

▶ **Theorem 4.10.** *Let* $\mathbf{G}$ *be a global type and* $\{\!\{H(\mathbf{G}){\Downarrow_{\mathtt{p}}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ *be its erasure candidate implementation. If* $\mathcal{L}_{\text{fin}}(\mathbf{G})$ *is implementable*[4]*, then* $\{\!\{H(\mathbf{G}){\Downarrow_{\mathtt{p}}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ *is deadlock free and* $\mathcal{L}_{\text{fin}}(\{\!\{H(\mathbf{G}){\Downarrow_{\mathtt{p}}}\}\!\}_{\mathtt{p}\in\mathcal{P}}) = \mathcal{L}_{\text{fin}}(\mathbf{G})$.

The proof can be found in Appendix C.3. This result does only account for finite languages so we extend it for infinite sequences.

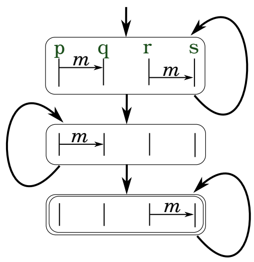
▶ **Lemma 4.11** (Erasure candidate implementation generalises to infinite language if implementable). *Let* $\mathbf{G}$ *be a* 0*-reachable global type and* $\{\!\{H(\mathbf{G}){\Downarrow_{\mathtt{p}}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ *be its erasure candidate implementation. If* $\mathbf{G}$ *is implementable, then* $\mathcal{L}_{\text{inf}}(\{\!\{H(\mathbf{G}){\Downarrow_{\mathtt{p}}}\}\!\}_{\mathtt{p}\in\mathcal{P}}) = \mathcal{L}_{\text{inf}}(\mathbf{G})$.

The proof can be found in Appendix C.4.

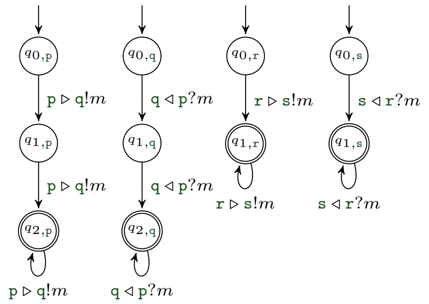
So far, we have shown that, if $\mathbf{G}$ is implementable, its erasure candidate implementation implements it. For this, we actually took the detour and showed the same for $H(\mathbf{G})$, the HMSC encoding of $\mathbf{G}$. For HMSCs, this is undecidable in general [63]. We show that, because of their conditions on choice, global types fall into the class of globally-cooperative HMSCs for which implementability is decidable.

▶ **Definition 4.12** (Communication graph [39]). *Let* $M = (N, p, f, l, (\leq_{\mathtt{p}})_{\mathtt{p}\in\mathcal{P}})$ *be an MSC. The* communication graph *of* M *is a directed graph with node* $\mathtt{p}$ *for every role* $\mathtt{p}$ *that sends or receives a message in* M *and edges* $\mathtt{p} \to \mathtt{q}$ *if* M *contains a message from* $\mathtt{p}$ *to* $\mathtt{q}$*, i.e., there is* $e \in N$ *such that* $p(e) = \mathtt{p}$ *and* $p(f(e)) = \mathtt{q}$.

[4] *Implementability* is lifted to languages as expected.



(a) Implementable HMSC H_{ing} that is not globally cooperative



(b) An implementation for H_{ing}

Figure 6 An implementable HMSC which is not globally-cooperative with its implementation

It is important that the communication graph of M does not have a node for every role but only the active ones, i.e., that send or receive in M.

▶ **Definition 4.13** (Globally-cooperative HMSCs [39]). *An HMSC $H = (V, E, v^I, V^T, \mu)$ is called* globally-cooperative *if for every loop, i.e., $v_1, \ldots, v_n$ with $(v_i, v_{i+1}) \in E$ for every $1 \leq i < n$ and $(v_n, v_1) \in E$, the communication graph of $\mu(v_1) \ldots \mu(v_n)$ is weakly connected.*[5]

We can check this directly on for a global type $\mathbf{G}$. It is straightforward to define a communication graph for words from Σ_{sync}^*. We check it on $\mathsf{GAut}(\mathbf{G})$: for each binder state, we check the communication graph for the shortest trace to every corresponding recursion state.

▶ **Theorem 4.14** (Thm. 3.7 [63]). *Let H be a globally-cooperative HMSC. Restricted to its finite language $\mathcal{L}_{\text{fin}}(H)$, safe realisability is* EXPSPACE*-complete.*

▶ **Lemma 4.15.** *Let $\mathbf{G}$ be an implementable 0-reachable global type. Then, its HMSC encoding $H(\mathbf{G})$ is globally-cooperative.*

The proof can be found in Appendix C.5 and is far from trivial. We explain the main intuition for the proof with the following example where we exemplify why the same result does not hold for HMSCs in general.

▶ **Example 4.16** (Implementable HMSC that is not globally cooperative). Let us consider the HMSC H_{ing} in Figure 6a. It is implementable but not globally-cooperative and not representable with a global type. This protocol consists of three loops. In the first one, `p` sends a message m to `q` while `r` sends a message m to `s`. This is the loop for which the communication graph is not weakly connected. In the second one, only the interaction between `p` and `q` is specified, while, in the third one, it is only the one between `r` and `s`. For a protocol, which consists of the first and third loop only, an implementation can always expose an execution with more interactions between `p` and `q` than the ones between `r` and `s` due to the lack of synchronisation. Here, the additional second loop can make up for such executions so any execution has a path that specifies it. Thus, this protocol can be implemented with the CSM built from the FSMs illustrated in Figure 6b. In Appendix C.6, we explain in detail why there is a path in H_{ing} for any trace of the CSM and how to modify the example not to have final states with outgoing transitions.

[5] Weakly connected means that, when considering every edge not to be directed, every node is connected with every other node.

▶ **Theorem 4.17.** *Let* $\mathbf{G}$ *be a* 0*-reachable global type with generalised choice. Checking implementability of* $\mathbf{G}$ *is in* EXPSPACE*.*

Proof. We construct $H(\mathbf{G})$ from $\mathbf{G}$ and check if it is globally-cooperative. For this, we apply the coNP-algorithm by Genest et al. [39] which is based on guessing a subgraph and checking its communication graph. If $H(\mathbf{G})$ is not globally cooperative, we know from Lemma 4.15 that $\mathbf{G}$ is not implementable. If $H(\mathbf{G})$ is globally cooperative, we check safe realisability for $H(\mathbf{G})$. By Theorem 4.14, this is in EXPSPACE. If $H(\mathbf{G})$ is not safely realisable, it trivially follows that $\mathbf{G}$ is not implementable. If $H(\mathbf{G})$ is safely realisable, $\mathbf{G}$ is implementable by the erasure candidate implementation with Theorem 4.10 and Lemma 4.11. ◀

With this, the implementability problem for global types with generalised choice is decidable.

▶ **Corollary 4.18.** *Let* $\mathbf{G}$ *be a* 0*-reachable global type with generalised choice. It is decidable whether* $\mathbf{G}$ *is implementable and there is an algorithm to obtain its implementation.*

Our results also apply to the stronger notion of progress (Remark 2.14). This also entails that any sent message can eventually be received in an implementation — a property sometimes called *eventual reception* [61, Def. 4]. This notion only asks for the possibility but we can ensure that no role starves in a non-final state during an infinite execution in two ways. First, we can impose a (strong) fairness assumption — as imposed by Castagna et al. [19]. Second, we can require that every loop branch contains at least all roles that occur in interactions of any path with which the protocol can finish.

The Odd Case of Infinite Loops Without Exits In practice, it is reasonable to assume a mechanism to terminate a protocol for maintenance for instance, justifying the 0-Reachable-Assumption (p.15). In theory, one can think of protocols for which it does not hold. They would simply recurse indefinitely and can never terminate. This allows interesting behaviour like two sets of roles that do not interact with each other as the following example shows.

▶ **Example 4.19.** Consider the following global type: $\mathbf{G} = \mu t.\, \mathtt{p} \to \mathtt{q} : m.\, \mathtt{r} \to \mathtt{s} : m.\, t$. This global type is basically the protocol which consists only of the first loop of H_{ing} from Example 4.16. It describes an infinite execution with two pairs of roles that send and receive messages independently. While this can be implemented for an infinite setting, such a loop could never be exited since the set of roles would need to synchronise on the number of times the loop was taken to satisfy language equality.

Expressiveness of Local Types Local types, like their global counterparts, have a distinct expression for termination: 0. Thus, if one considers the FSM of a local type, every final state has no outgoing transitions. Our proposed algorithm might produce state machines for which this is not true. However, the language of such a state machine cannot be represented as local type. Both, our construction and local types are deterministic. Thus, if there is a final state with an outgoing transition, there cannot be any state machine that only has final states without outgoing transitions.

In addition, the syntax prescribes the structure of the state machines similarly as for global types: state machines for local types are also ancestor-recursive, free of intermediate recursion, non-merging and dense (Proposition 3.6). We believe that this is rather a result of the classical projection operator than a design choice. For our algorithm, this is not the case. This raises two obvious directions for future work. On the one hand, it might be feasible to find rewriting techniques that take arbitrary state machines without final states with outgoing transitions and transform them in a way such that they correspond to a local type. A naive approach to establish ancestor-recursiveness will most likely involve copying parts of the state machine. Such a rewriting would allow to re-use existing work, e.g., on sub-typing, which intuitively attempts to give freedom to implementations while preserving the soundness properties, On the other hand, one could also waive the syntactic restrictions and study sub-typing for this potentially more general class of local specifications.

On Lower Bounds for Implementability For general globally-cooperative HMSCs, i.e., that are not necessary the encoding of a global type, safe realisability is EXPSPACE-hard [63]. This hardness result does not carry over for $H(\mathbf{G})$ of a global type $\mathbf{G}$. The construction exploits that HMSCs do not impose any restrictions on choice. Global types, however, require every branch to be chosen by a single sender.

5 MSC Techniques for MST Verification

In the previous section, we generalised results from the MSC literature to show decidability of the implementability problem for global types from MSTs. However, the resulting algorithm suffers from high complexity. This is also true for the original problem of safe realisability of HMSCs. In fact, the problem is undecidable for general HMSCs. Besides globally-cooperative HMSCs, further restrictions of HMSCs have been studied to obtain algorithms with better complexity for global types. The results from the previous section, in particular Theorem 4.10 and Lemma 4.11, make most of these results applicable to the MST setting. One solely needs to check that the global type (or its HMSC encoding) belongs to the respective class. First, we transfer the algorithms for $\mathcal{I}$-closed HMSCs, which requires an HMSC not to exhibit certain anti-patterns of communication, to global types. Second, we explain approaches for HMSCs that introduced the idea of choice to HMSCs and a characterisation of implementable MSC languages. These can be a reasonable starting point for the design of complete algorithms for the implementability problem with better worst-case complexity. Third, we present a variant of the implementability problem. It can make unimplementable global types implementable without changing a protocol's structure but also help if the complexity of the previous algorithms is intractable. From now on, we may use the term implementability for HMSCs instead of safe realisability.

$\mathcal{I}$-closed Global Types

For globally-cooperative HMSCs, the implementability problem is EXPSPACE-complete. The membership in EXPSPACE was shown by reducing the problem to implementability of $\mathcal{I}$-closed HMSCs [63, Thm. 3.7]. These require the language of an HMSC to be closed with regard to an independence relation $\mathcal{I}$, where, intuitively, two interactions are independent if there is no role which is involved in both. Implementability for $\mathcal{I}$-closed HMSCs is PSPACE-complete [63, Thm. 3.6]. As for the EXPSPACE-hardness for globally-cooperative HMSCs, the PSPACE-hardness exploits features that cannot be modelled with global types and there might be algorithms with better worst-case complexity.

We adapt the definitions [63] to the MST setting. These consider atomic BMSCs, which are BMSCs that cannot be split further. With the HMSC encoding for global types, it is straightforward that atomic BMSCs correspond to individual interactions for global types. Thus, we define the independence relation $\mathcal{I}$ on the alphabet Σ_{sync}.

▶ **Definition 5.1** (Independence relation $\mathcal{I}$). *We define the independence relation $\mathcal{I}$ on Σ_{sync}:*

$$\mathcal{I} := \{(\mathtt{p}\to\mathtt{q}:m, \mathtt{r}\to\mathtt{s}:m') \mid \{\mathtt{p},\mathtt{q}\} \cap \{\mathtt{r},\mathtt{s}\} \neq \emptyset)\}$$

We lift this to an equivalence relation $\equiv_{\mathcal{I}}$ on words as its transitive and reflexive closure:

$$\equiv_{\mathcal{I}} := \{(u.\, x_1.\, x_2.\, w, u.\, x_2.\, x_1.\, w) \mid u, w \in \Sigma^*_{sync} \text{ and } (x_1, x_2) \in \mathcal{I}\}$$

We define its closure for language $L \subseteq \Sigma^*_{sync}$: $\mathcal{C}^{\equiv_{\mathcal{I}}}(L) := \{u \in \Sigma^*_{sync} \mid \exists w \in L \text{ with } u \equiv_{\mathcal{I}} w\}$.

► **Definition 5.2** ($\mathcal{I}$-closedness for global types)**.** *Let* $\mathbf{G}$ *be a global type* $\mathbf{G}$*. We say* $\mathbf{G}$ *is* $\mathcal{I}$*-closed if* $\mathcal{L}_{\text{fin}}(\mathsf{GAut}(\mathbf{G})) = \mathcal{C}^{\equiv_{\mathcal{I}}}(\mathcal{L}_{\text{fin}}(\mathsf{GAut}(\mathbf{G})))$*.*

Note that $\mathcal{I}$-closedness is defined on the state machine $\mathsf{GAut}(\mathbf{G})$ of $\mathbf{G}$ with alphabet Σ_{sync} and not on its semantics $\mathcal{L}(\mathbf{G})$ with alphabet Σ_{async}.

► **Example 5.3.** The global type $\mathbf{G}_{2\text{BP}}$ is $\mathcal{I}$-closed. Buyer $\mathtt{a}$ is involved in every interaction. Thus, for every consecutive interactions, there is a role that is involved in both.

► **Algorithm 1** (Checking if $\mathbf{G}$ is $\mathcal{I}$-closed)**.** *Let* $\mathbf{G}$ *be a global type* $\mathbf{G}$*. We construct the state machine* $\mathsf{GAut}(\mathbf{G})$*. We need to check every consecutive occurrence of elements from* Σ_{sync} *for words from* $\mathcal{L}(\mathsf{GAut}(\mathbf{G}))$*. For binder states, incoming and outgoing transition labels are always* ε*. This is why we slightly modify the state machine but preserve its language. We remove all variable states and rebend their only incoming transition to the state their only outgoing transition leads to. In addition, we merge binder states with their only successor. For every state* q *of this modified state machine, we consider the labels* $x, y \in \Sigma_{sync}$ *of every combination of incoming and outgoing transition of* q*. We check if* $x \equiv_{\mathcal{I}} y$*. If this is true for all* x *and* y*, we return* true*. If not, we return* false*.*

► **Lemma 5.4.** *A global type* $\mathbf{G}$ *is* $\mathcal{I}$*-closed iff Algorithm 1 returns* true*.*

The proof can be found in Appendix D. This shows that the presented algorithm can be used to check $\mathcal{I}$-closedness. The algorithm considers every state and all combinations of transitions leading to and from it.

► **Proposition 5.5.** *For global type* $\mathbf{G}$*, checking* $\mathcal{I}$*-closedness of* $\mathbf{G}$ *is in* $O(|\mathbf{G}|^2)$*.*

The tree-like shape of $\mathsf{GAut}(\mathbf{G})$ might suggest that this check can be done in linear time. However, the following example shows that recursion can lead to a quadratic number of checks.

► **Example 5.6.** Consider the following global type for some n:

$$\mu t.\; + \begin{cases} \mathtt{p}\to\mathtt{q}_0 : m_0.\, \mathtt{q}_0 \to \mathtt{r}_0 : m_0.\, \mathtt{r}_0 \to \mathtt{s}_0 : m_0.\, 0 \\ \mathtt{p}\to\mathtt{q}_1 : m_1.\, \mathtt{q}_1 \to \mathtt{r}_1 : m_1.\, \mathtt{r}_1 \to \mathtt{s}_1 : m_1.\, t \\ \quad\vdots \\ \mathtt{p}\to\mathtt{q}_n : m_n.\, \mathtt{q}_n \to \mathtt{r}_n : m_n.\, \mathtt{r}_n \to \mathtt{s}_n : m_n.\, t \end{cases} .$$

It is obvious that $(\mathtt{p}\to\mathtt{q}_i : m_i, \mathtt{q}_i \to \mathtt{r}_i : m_i) \notin \mathcal{I}$ and $(\mathtt{q}_i \to \mathtt{r}_i : m_i, \mathtt{r}_i \to \mathtt{s}_i : m_i) \notin \mathcal{I}$ for every i. Because of the recursion, we need to check if $(\mathtt{r}_i \to \mathtt{s}_i : m_i, \mathtt{p} \to \mathtt{q}_j : m_j)$ is in $\mathcal{I}$ for every $0 \neq i \neq j$. This might lead to a quadratic number of checks.

If a global type $\mathbf{G}$ is $\mathcal{I}$-closed, we can apply the respective results for its HMSC $H(\mathbf{G})$ and the resulting CSM is also an implementation for $\mathbf{G}$. If not, we need to consider other approaches — where the last resort are the algorithms for globally-cooperative HMSCs. There are global types that are not $\mathcal{I}$-closed but implementable.

► **Example 5.7.** The following implementable global type is not $\mathcal{I}$-closed: $\mathtt{p}\to\mathtt{q}:m.\, \mathtt{r}\to\mathtt{s}:m.\, 0$.

Detecting Non-local Choice in HMSCs

For HMSCs, there is no restriction on branching. Similar to choice for global types, the idea of imposing restrictions on choice was studied for HMSCs [10, 68, 66, 44, 39]. We refer to Section 7 for an overview. Here, we focus on results that seem most promising for developing algorithms to check implementability of global types with better worst-case complexity. The work by Dan et al. [32] centers around the idea of non-local choice. Intuitively, non-local choice yields scenarios which makes it impossible to implement the language. In fact, if a language is not implementable, there is some non-local choice. Thus, checking implementability amounts to checking non-local choice freedom. For this definition, they showed insufficiency of Baker's condition [7] and reformulated the closure conditions for safe realisability by Alur et al. [3]. In particular, they provide a definition which is based on projected words of a language in contrast to explicit choice. While it is straightforward to check their definition for finite collections of k BMSCs with n events in $O(k^2 \cdot |\mathcal{P}| + n \cdot |\mathcal{P}|)$, it is unclear how to check their condition for languages with infinitely many elements. The design of such a check is far from trivial as their definition does not give any insight about local behaviour and their algorithm heavily relies on the finite nature of finite collections of BMSCs. Still, we believe that the observations based on the closure conditions by Alur et al. [3], which provide a sound and complete characterisation of implementable languages, can be key to more efficient complete algorithms for the implementability problem for global types from MSTs.

Payload Implementability

A deadlock free CSM implements a global type if their languages are precisely the same. In the HMSC domain, a variant of the implementability problem has been studied. Intuitively, it allows to add fresh data to the payload of an existing message and protocol fidelity allows to omit the additional payload data. This allows to add synchronisation messages to existing interactions and can make unimplementable global types implementable without changing the structure of the protocol. It can also be used if a global type is rejected by projection and the run time of the previous algorithms is not acceptable.

▶ **Definition 5.8** (Payload implementability). *Let L be a language with message alphabet $\mathcal{V}_1$. We say that L is* payload implementable *if there is a message alphabet $\mathcal{V}_2$ for a deadlock free CSM $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ with $A_\mathtt{p}$ over $\{\mathtt{p}\triangleright\mathtt{q}!m, \mathtt{p}\triangleleft\mathtt{q}?m \mid \mathtt{q}\in\mathcal{P},\ m\in\mathcal{V}_1\times\mathcal{V}_2\}$ such that its language is the same when projecting on to the message alphabet $\mathcal{V}_1$, i.e., $\mathcal{C}^\sim(L) = \mathcal{L}(\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}})\Downarrow_{\mathcal{V}_1}$, where $(\mathtt{p}\triangleright\mathtt{q}!(m_1,m_2))\Downarrow_{\mathcal{V}_1} := \mathtt{p}\triangleright\mathtt{q}!m_1$ and $(\mathtt{q}\triangleleft\mathtt{p}?(m_1,m_2))\Downarrow_{\mathcal{V}_1} := \mathtt{q}\triangleleft\mathtt{p}?m_1$ and is lifted to words and languages as expected.*

Genest et al. [39] identified a class of HMSCs which is always payload implementable with a deadlock free CSM of linear size.

▶ **Definition 5.9** (Local HMSCs [39]). *Let $H = (V, E, v^I, V^T, \mu)$ be an HMSC. We say that H is* local *if $\mu(v^I)$ has a unique minimal event and there is a function* root: $V \to \mathcal{P}$ *such that for every $(v,u)\in E$, it holds that $\mu(u)$ has a unique minimal event e and e belongs to* $\mathrm{root}(v)$*, i.e., for $\mu(u) = (N, p, f, l, (\leq_\mathtt{p})_{\mathtt{p}\in\mathcal{P}})$, we have that $p(e) = \mathrm{root}(v)$ and $e \leq e'$ for every $e' \in N$.*

▶ **Proposition 5.10** (Prop. 21 [39]). *For any local HMSC H, $\mathcal{L}_{\mathrm{fin}}(H)$ is payload implementable.*

The algorithm to construct a deadlock free CSM [39, Sec. 5.2] suggests that the BMSCs for such HMSCs need to be maximal – in the sense that any vertex with a single successor is collapsed with its successor. If this was not the case, the result would claim that the language of the following global type is payload implementable: $\mu t. + \begin{cases} \mathtt{p} \to \mathtt{q} : m_1 . \mathtt{r} \to \mathtt{s} : m_2 . t \\ \mathtt{p} \to \mathtt{q} : m_3 . \end{cases}$. However, is is easy to see that it is not payload implementable since there is no interaction between $\mathtt{p}$, which decides whether to stay in the loop or not, and $\mathtt{r}$. Thus, we cannot simply check whether $H(\mathbf{G})$ is local. In fact, it would always be. Instead, we first need to minimise it and then check whether it is local. If we collapse the two consecutive vertices with independent pairs of roles in this example, the HMSC is not local. The representation of the HMSC matters which shows that local as property is rather a syntactic than a semantic notion.

▶ **Algorithm 2** (Checking if $\mathbf{G}$ is *local*). *Let $\mathbf{G}$ be a global type $\mathbf{G}$. We consider the finite trace w' of every longest branch-free, loop-free and non-initial run in the state machine* $\mathsf{GAut}(\mathbf{G})$. *We split the (synchronous) interactions into asynchronous events: $w = \mathrm{split}(w') = w_1 \dots w_n$. We need to check if there is $u \sim w$ with $u = u_1 \dots u_n$ such that $u_1 \neq w_1$. For this, we can construct an MSC for w' [38, Sec. 3.1] and check if there is a single minimal event, because MSCs are closed under $\approx$ [77, Lm. 5]. If this is the case for any trace w', we return* false. *If not, we return* true.

It is straightforward that this mimics the corresponding check for the HMSC $H(\mathbf{G})$ and, including similar modifications as for Algorithm 1, the check can be done in $O(|\mathbf{G}|)$.

▶ **Proposition 5.11.** *For a global type $\mathbf{G}$, Algorithm 2 returns* true *iff $H(\mathbf{G})$ is local.*

Ben-Abdallah and Leue [10] introduced local-choice HMSCs which are as expressive as local HMSCs. Their condition also uses a root-function and minimal events but quantifies over paths. Every local HMSC is a local-choice HMSC and every local-choice HMSC can be translated to a local HMSC, which accepts the same language, with a quadratic blow-up [39]. It is straightforward to adapt the Algorithm 2 to check if a global type is local-choice. If this is the case, we translate the protocol and use the implementation for the translated protocol.

6 Implementability with Intra-role Reordering

In this section, we introduce a generalisation of the implementability problem that relaxes the total event order for each role and allows to reorder receive events. We prove that this generalisation is undecidable in general.

A Case for More Reordering

From the perspective of a single role, each word in its language consists of a sequence of receive and send events. Choice in global types happens by sending (and not by receiving). Because of this, one can argue that a role should be able to receive messages from different senders in any order between sending two messages. In practice, receiving a message can induce a task with non-trivial computation, which is not reflected in our model. Thus, such a reordering for a sequence of receive events can have outsized performance benefits. In addition, there are global types that can be implemented with regard to this generalised relation even if no (standard) implementation exists.

▶ **Example 6.1** (Example for intra-role reordering). Let us consider a global type where a central coordinator $\mathtt{p}$ distributes independent tasks to different roles in rounds:

$$\mathbf{G}_{\mathrm{TC}} := \mu t. \begin{cases} \mathtt{p} \to \mathtt{q}_1 : \mathrm{task} \,.\, \dots \mathtt{p} \to \mathtt{q}_n : \mathrm{task} \,.\, \mathtt{q}_1 \to \mathtt{p} : \mathrm{result} \,.\, \dots \mathtt{q}_n \to \mathtt{p} : \mathrm{result} \,.\, t \\ \mathtt{p} \to \mathtt{q}_1 : \mathrm{done} \,.\, \dots \mathtt{p} \to \mathtt{q}_n : \mathrm{done} \,.\, 0 \end{cases}.$$

Since all tasks in each round are independent, $\mathtt{p}$ can benefit from receiving the results in the order they arrive instead of busy-waiting.

We generalise the indistinguishability relation $\sim$ accordingly.

▶ **Definition 6.2** (Intra-role indistinguishability relation). *We define a family of* intra-role indistinguishability relations $\approx_i \subseteq \Sigma^* \times \Sigma^*$, *for* $i \geq 0$ *as follows. For all* $w, u \in \Sigma^*$, $w \sim_i u$ *entails* $w \approx_i u$. *For* $i = 1$, *we define: if* $\mathtt{q} \neq \mathtt{r}$, *then* $w.\mathtt{p}\triangleleft\mathtt{q}?m.\mathtt{p}\triangleleft\mathtt{r}?m'.u \ \sim_1 \ w.\mathtt{p}\triangleleft\mathtt{r}?m'.\mathtt{p}\triangleleft\mathtt{q}?m.u$. *Based on this, we define* $\approx$ *analogously to* $\sim$. *Let* w, w', w'' *be words s.t.* $w \approx_1 w'$ *and* $w' \approx_i w''$ *for some* i. *Then* $w \approx_{i+1} w''$. *We define* $w \approx u$ *if* $w \approx_n u$ *for some* n. *It is straightforward that* $\approx$ *is an equivalence relation. Define* $u \preceq_\approx v$ *if there is* $w \in \Sigma^*$ *such that* $u.w \approx v$. *Observe that* $u \sim v$ *iff* $u \preceq_\approx v$ *and* $v \preceq_\approx u$. *We extend* $\approx$ *to infinite words and languages as for* $\sim$.

▶ **Definition 6.3** (Implementability with intra-role reordering $\approx$). *A global type* $\mathbf{G}$ *is said to be* implementable with regard to $\approx$ *if there exists a deadlock free CSM* $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ *such that (i)* $\mathcal{L}(\mathbf{G}) \subseteq \mathcal{C}^\approx(\mathcal{L}(\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}))$ *and (ii)* $\mathcal{C}^\approx(\mathcal{L}(\mathbf{G})) = \mathcal{C}^\approx(\mathcal{L}(\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}))$. *We say that* $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ $\approx$*-implements* $\mathbf{G}$.

In this section, we emphasise the indistinguishability relation, e.g., $\approx$-implementable, which is considered. We could have also followed the definition of $\sim$-implementability and required $\mathcal{C}^\approx(\mathcal{L}(\mathbf{G})) = \mathcal{L}(\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}})$. This, however, requires the CSM to be closed under $\approx$. In general, this might not be possible with a finite number of states. In particular, if there is a loop without send event for a role, the labels in the loop would introduce an infinite closure if we require that $\mathcal{C}^\approx(\mathcal{L}(\mathbf{G}))\Downarrow_{\Sigma_\mathtt{r}} = \mathcal{L}(A_\mathtt{r})$.

▶ **Example 6.4.** We consider a variant of $\mathbf{G}_{\mathrm{TC}}$ from Example 6.1 with $n = 2$ where $\mathtt{q}_1$ and $\mathtt{q}_2$ send a log message to $\mathtt{r}$ after receiving the task and before sending the result back:

$$\mathbf{G}_{\mathrm{TCLog}} := \mu t. \begin{cases} \mathtt{p}\to\mathtt{q}_1 : \mathrm{task} \,.\, \mathtt{p}\to\mathtt{q}_2 : \mathrm{task} \,.\, \mathtt{q}_1\to\mathtt{r} : \mathrm{log} \,.\, \mathtt{q}_2\to\mathtt{r} : \mathrm{log} \,.\, \mathtt{q}_1\to\mathtt{p} : \mathrm{result} \,.\, \mathtt{q}_2\to\mathtt{p} : \mathrm{result} \,.\, t \\ \mathtt{p}\to\mathtt{q}_1 : \mathrm{done} \,.\, \mathtt{p}\to\mathtt{q}_2 : \mathrm{done} \,.\, 0 \end{cases} .$$

There is no FSM for $\mathtt{r}$ that precisely accepts $\mathcal{C}^\approx(\mathcal{L}(\mathbf{G}_{\mathrm{TCLog}}))\Downarrow_{\Sigma_\mathtt{r}}$ as it would need keep count of the difference at any point in time which can be unbounded. If we rely on the fact that $\mathtt{q}_1$ and $\mathtt{q}_2$ send the same number of log-messages to $\mathtt{r}$, we can use an FSM $A_\mathtt{r}$ with a single state (both initial and final) with two transitions: one for the log-message from $\mathtt{q}_1$ and $\mathtt{q}_2$ each, that lead back to the same state. For this, it holds that $\mathcal{C}^\approx(\mathcal{L}(\mathbf{G}_{\mathrm{TCLog}}))\Downarrow_{\Sigma_\mathtt{r}} \subseteq \mathcal{L}(A_\mathtt{r})$.

This is why we chose a more permissive definition which is required to cover at least as much as specified in the global type (i) and the $\approx$-closure of both are the same (ii).

It is trivial that any $\sim$-implementation for a global type does also $\approx$-implement it.

▶ **Proposition 6.5.** *Let* $\mathbf{G}$ *be a global type that is* $\sim$*-implemented by the CSM* $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$. *Then,* $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ *also* $\approx$*-implements* $\mathbf{G}$.

For instance, the task coordination protocol from Example 6.1 can be $\sim$-implemented as well as $\approx$-implemented by an erasure candidate implementation. Still, $\approx$-implementability gives more freedom and allows to consider all possible combinations of arrivals of results. In addition, $\approx$-implementability renders some global types implementable which would not be otherwise. For instance, those with a role that would need to receive different sequences, which are related by $\approx$, in different branches it cannot distinguish (yet).

▶ **Example 6.6** ($\approx$-implementable but not $\sim$-implementable). Let us consider the following global type: $(\mathtt{p}\to\mathtt{q} : l.\,\mathtt{p}\to\mathtt{r} : m.\,\mathtt{q}\to\mathtt{r} : m.\,0) + (\mathtt{p}\to\mathtt{q} : r.\,\mathtt{q}\to\mathtt{r} : m.\,\mathtt{p}\to\mathtt{r} : m.\,0)$. This cannot be $\sim$-implemented because $\mathtt{r}$ would need to know the branch to receive the messages from $\mathtt{p}$ and $\mathtt{q}$ in the correct order. However, it is $\approx$-implementable. The FSMs for $\mathtt{p}$ and $\mathtt{q}$ can be obtained with projection by erasure. For $\mathtt{r}$, we can have an FSM that only accepts $\mathtt{r}\triangleleft\mathtt{p}?m.\,\mathtt{r}\triangleleft\mathtt{q}?m$ but also an FSM which accepts $\mathtt{r}\triangleleft\mathtt{q}?m.\,\mathtt{r}\triangleleft\mathtt{p}?m$ in addition. Note that $\mathtt{r}$ does not learn the choice in the second FSM even if it branches. Hence, it would not be

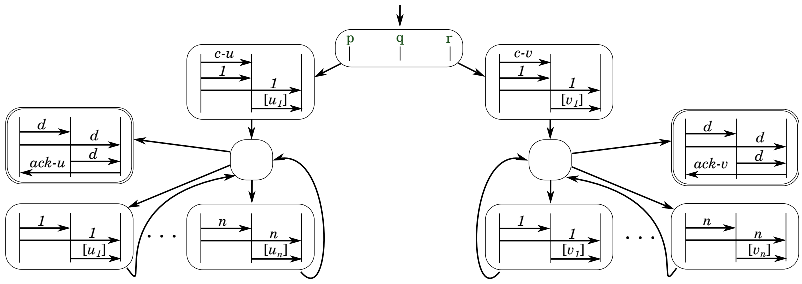


Figure 7 HMSC encoding $H(\mathbf{G}_{\text{MPCP}})$ of the MPCP encoding

implementable if it sent different messages in both branches later on. However, it could still learn by receiving and, afterwards, send different messages.

Implementability with Intra-role Reordering is Undecidable

Unfortunately, checking implementability with regard to $\approx$ for global types (with directed choice) is undecidable. Intuitively, the reordering allows roles to drift arbitrarily far apart as the execution progresses which makes it hard to learn which choices were made.

We reduce the *Post Correspondence Problem* (PCP) [73] to the problem of checking implementability with regard to $\approx$. An instance of PCP over an alphabet Δ, $|\Delta| > 1$, is given by two finite lists $(u_1, u_2, \ldots, u_n)$ and $(v_1, v_2, \ldots, v_n)$ of finite words over Δ. A solution to the instance is a sequence of indices $(i_j)_{1 \leq j \leq k}$ with $k \geq 1$ and $1 \leq i_j \leq n$ for all $1 \leq j \leq k$, such that $u_{i_1} \ldots u_{i_k} = v_{i_1} \ldots v_{i_k}$. To be precise, we present a reduction from the modified PCP (MPCP) [76, Sec. 5.2], which is also undecidable. It simply requires that a match starts with a specific pair — in our case we choose the pair with index 1. It is possible to directly reduce from PCP but the reduction of MPCP is more concise. Intuitively, we require that the solution starts with the first pair so there exists no trivial solution and choosing a single pair is more concise than all possible ones. Our encoding is the following global type where $x \in \{u, v\}$, $[x_i]$ denotes a sequence of message interactions with message $x_i[1], \ldots, x_i[k]$ each for x_i of length k, message $c\text{-}x$ indicates choosing tile set x, and message $ack\text{-}x$ indicates acknowledging the tile set x:

$$
\begin{aligned}
\mathbf{G}_{\text{MPCP}} \quad &:= \quad + \begin{cases} G(u, \mathtt{r}\to\mathtt{p}: ack\text{-}u.\, 0) \\ G(v, \mathtt{r}\to\mathtt{p}: ack\text{-}v.\, 0) \end{cases} \quad \text{with} \\
G(x, X) \quad &:= \quad \mathtt{p}\to\mathtt{q}: c\text{-}x.\, \mathtt{p}\to\mathtt{q}: 1.\, \mathtt{p}\to\mathtt{r}: 1.\, \mathtt{q}\to\mathtt{r}: [x_1].\, \mu t_1.\, + \begin{cases} \mathtt{p}\to\mathtt{q}: 1.\, \mathtt{p}\to\mathtt{r}: 1.\, \mathtt{q}\to\mathtt{r}: [x_1].\, t_1 \\ \quad \vdots \\ \mathtt{p}\to\mathtt{q}: n.\, \mathtt{p}\to\mathtt{r}: n.\, \mathtt{q}\to\mathtt{r}: [x_n].\, t_1 \\ \mathtt{p}\to\mathtt{q}: d.\, \mathtt{p}\to\mathtt{r}: d.\, \mathtt{q}\to\mathtt{r}: d.\, X \end{cases} .
\end{aligned}
$$

The HMSC encoding $H(\mathbf{G}_{\text{MPCP}})$ is illustrated in Figure 7. Intuitively, $\mathtt{r}$ eventually needs to know which branch was taken in order to match $ack\text{-}x$ with $c\text{-}x$ from the beginning. However, it can only know if there is no solution to the MPCP instance. In the full proof in Appendix E, we show that $\mathbf{G}_{\text{MPCP}}$ is $\approx$-implementable iff the MPCP instance has no solution.

▶ **Theorem 6.7.** *Checking implementability with regard to $\approx$ for global types with directed choice is undecidable.*

This result carries over to HMSCs if we consider safe realisability with regard to $\approx$.

▶ **Definition 6.8** (Safe realisability with regard to $\approx$). *An HMSC H is said to be* safely realisable with regard to $\approx$ *if there exists a deadlock-free CSM $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ such that the following holds: (i) $\mathcal{L}(H) \subseteq \mathcal{C}^{\approx}(\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}))$ and (ii) $\mathcal{C}^{\approx}(\mathcal{L}(H)) = \mathcal{C}^{\approx}(\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}))$.*

▶ **Corollary 6.9.** *Checking safe realisability with regard to $\approx$ for HMSCs is undecidable.*

In fact, the HMSC encoding for $\mathbf{G}_{\text{MPCP}}$ satisfies a number of channel restrictions. The HMSC $H(\mathbf{G}_{\text{MPCP}})$ is existentially 1-bounded, 1-synchronisable and half-duplex [77]. For details on these channel restrictions, we refer to work by Stutz and Zufferey [77, Sec. 3.1].

▶ Remark 6.10 (Sending to the rescue). The MPCP encoding only works since receive events can be reordered unboundedly in an execution. If we amended the definition of $\approx$ to give each receive event a budget that depletes with every reordering, this encoding would not be possible. Alternatively, one could require every active role in a loop to send at least once. This also prevents such an unbounded reordering. For such restrictions on the considered indistinguishability relation, the corresponding implementability problem likely becomes decidable. We leave a detailed analysis for future work.

7 Related Work

In this section, we solely cover related work which we have not discussed before.
Session types originate in process algebra and were first introduced by Honda et al. [46] for binary session. For systems with more than two roles, they have been extended to multiparty session types [48]. Their connection to linear logic [41] has been studied subsequently [37, 84, 17]. In this work, we explained MST frameworks with classical projection operators. Other approaches do not focus on projection but, for instance, employ model checking [75] or only apply ideas from MST without the need for global types [61].

Our decidability result applies to global types with generalised choice. There are only few MST frameworks that effectively allow to generate local types from global types with generalised choice for an asynchronous setting. Castellani et al. [20] consider a synchronous setting. The same holds for the work by Jongmans and Yoshida [55] but their parallel operator allows to model some asynchrony with bag semantics. The setting in the work by Lange et al. [59] yields semantics similar to Petri nets. To the best of our knowledge, the work by Castagna et al. [19] is the only one to attempt completeness for global types with generalised choice. However, their notion of completeness allows to omit redundant executions for underspecified global types [19, Def. 4.1]. Their conditions, given as inference rules, are not effective and their algorithmically checkable conditions can only exploit local information to disambiguate choices. In contrast, Majumdar et al. [64] employ a global availability analysis but, as classical projection operator, it suffers from the shortcomings presented in this work. For a detailed overview of frameworks allowing generalised choice, we refer to their work [64]. They also present a counterexample to the implementability conditions formulated for Choreography Automata [8]. The global types by Dagnino et al. [31] specify send and receive events independently but each term requires to send to a single receiver and to receive from a single sender upon branching. They present a sound and complete type inference algorithm that infers all global types for a given system.

Here, we do not distinguish between local types and implementations but use the local types directly as implementations. Intuitively, subtyping studies possibilities to give freedom in the implementation while preserving the soundness properties. The intra-role indistinguishability relation $\approx$, which allows to reorder receive events for a role, resembles subtyping to some extent, e.g., the work by Cutner et al. [30]. A detailed investigation of this relation is beyond the scope of this work. For details on subtyping, we refer to work by Chen et al. [27, 26], Lange and Yoshida [60], and Bravetti et al. [16].

Various extensions to make MST verification applicable to more scenarios were studied: for instance delegation [47, 48, 21], dependent session types [80, 35, 81], parametrised session types [24, 35], gradual session types [51], or dynamic self-adaption [43]. Context-free session types [79, 56] provide a more expressive way to specify protocols in the MST domain. Recently, research on fault-tolerant MSTs [83, 9, 72] investigated ways to weaken the strict assumptions about reliable channels.

Choreographic programming [29, 40, 45] applies a similar approach as MSTs: they allow to specify a global protocol specification with joint send and receive events and project to end-point views. As for Pirouette by Hirsch and Garg [45], there are first mechanisations of MST frameworks [78, 22, 53, 52].

The connection of MSTs and CSMs was studied soon after MSTs had been proposed [34]. CSMs are known to be Turing-powerful [15]. Decidable classes have been obtained for different semantics, e.g., half-duplex communication for two roles [23], input-bounded [12], and unreliable/lossy channels [2], as well for restricted communication topology [71, 82]. Similar restrictions for CSMs are existential boundedness [38] and synchronisability [14, 42]. It was shown that global types can only express existentially 1-bounded, 1-synchronisable and half-duplex communication [77] while Bollig et al. [13] established a connection between synchronisability and MSO logic.

Globally-cooperative HMSCs were independently introduced by Morin [67] as c-HMSCs. Their communication graph is weakly connected. The class of bounded HMSCs [5] requires it to be strongly connected. Historically, it was introduced before the class of globally-cooperative HMSCs and, after the latter has been introduced, safe realisability for bounded HMSCs was also shown to be EXPSPACE-complete [63]. This class was independently introduced as regular HMSCs by Muscholl and Peled [69]. Both terms are justified: the language generated by a *regular* HMSC is regular and every *bounded* HMSC can be implemented with universally bounded channels. In fact, a HMSC is bounded if and only if it is a globally-cooperative and it has universally bounded channels [39, Prop. 4].

We cover approaches which introduced the idea of choice to HMSCs that were not discussed in Section 5. Ben-Abdallah and Leue [10] approached the realisability problem by defining and detecting non-local choice, which are basically choices not made by a single role. Their semantics, however, incorporates queuing behaviour that renders their systems finite-state. Another line of work [68, 66] identified that non-local choice and implied scenarios are strongly coupled. An implied scenario is an execution, which is not specified in the HMSC, but any candidate implementation necessarily exposes. Initial attempts by Muccini [68] yielded contradictory observations as shown by Mooij et al. [66] so they proposed variants of non-local choice but they accept the implied scenarios from such choices as given in the HMSC. Hélouët and Jard [44] pointed out that the absence of non-local choice does not guarantee implementability but just less ambiguity. They proposed reconstructibility which shall entail implementability. Majumdar et al. [64] showed that their notion of reconstructibility, with the requirement of unique messages, is quite restrictive but also flawed.

In addition to local HMSCs, Genest et al. [39] also introduced locally-cooperative HMSCs. Intuitively, they require for every two successors that each of their communication graphs as well as their concatenation's communication graph is weakly connected but it is only known that checking weak realisability (the one allowing deadlocks) has linear time complexity. Non-FIFO channel semantics has also been considered for HMSCs for which the complexity for safe realisability does not change while it has an influence on weak realisability [63].

8 Conclusion

We have proven decidability of the implementability problem for global types with generalised choice from MSTs — under the mild assumption that protocols can (but do not need to) terminate. To point at the origin for incompleteness of classical projection operators, we gave a visual explanation of the projection with various merge operators on finite state machines, which define the semantics of global and local types. To prove decidability, we formally related the implementability problem for global types with the safe realisability problem for HMSCs. While safe realisability is undecidable, we showed that implementable global types do always belong to the class of globally-cooperative HMSCs. There are global types that are outside of this class but the syntax of global types allowed us to prove that those can not be implemented. Another key was the extension of the HMSC results to infinite executions. We gave a comprehensive overview of MSC techniques and adapted some to the MST setting. Furthermore, we introduced a performance-oriented generalisation of the implementability problem which, however, we proved to be undecidable in general.

References

1. Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*, pages 165–175. IEEE Computer Society, 1988. doi:10.1109/LICS.1988.5115.
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. Inference of message sequence charts. *IEEE Trans. Software Eng.*, 29(7):623–633, 2003. doi:10.1109/TSE.2003.1214326.
4. 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.
5. Rajeev Alur and Mihalis Yannakakis. Model checking of message sequence charts. In Jos C. M. Baeten and Sjouke Mauw, editors, *CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24-27, 1999, Proceedings*, volume 1664 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 1999. doi:10.1007/3-540-48320-9_10.
6. 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.
7. 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.
8. 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.
9. Adam D. Barwell, Alceste Scalas, Nobuko Yoshida, and Fangyi Zhou. Generalised multiparty session types with crash-stop failures. In Bartek Klin, Slawomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland*, volume 243 of *LIPIcs*, pages 35:1–35:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CONCUR.2022.35.
10. 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.
11. 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, Proceed-*

ings, volume 11423 of *Lecture Notes in Computer Science*, pages 583–610. Springer, 2019. doi:10.1007/978-3-030-17184-1_21.

12 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.

13 Benedikt Bollig, Cinzia Di Giusto, Alain Finkel, Laetitia Laversa, Étienne Lozes, and Amrita Suresh. A unifying framework for deciding synchronizability. In Serge Haddad and Daniele Varacca, editors, *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference*, volume 203 of *LIPIcs*, pages 14:1–14:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CONCUR.2021.14.

14 Ahmed Bouajjani, Constantin Enea, Kailiang Ji, and Shaz Qadeer. On the completeness of verifying message passing programs under bounded asynchrony. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 372–391. Springer, 2018. doi:10.1007/978-3-319-96142-2_23.

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

16 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.

17 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.

18 Marco Carbone, Kohei Honda, N. Yoshida, R. Milner, G. Brown, and Steve Ross-Talbot. A theoretical basis of communication-centred concurrent programming. 2005.

19 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.

20 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.

21 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.

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

23 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.

24 Minas Charalambides, Peter Dinges, and Gul A. Agha. Parameterized, concurrent session types for asynchronous multi-actor interactions. *Sci. Comput. Program.*, 115-116:100–126, 2016. doi:10.1016/j.scico.2015.10.006.

25 Ruofei Chen, Stephanie Balzer, and Bernardo Toninho. Ferrite: A judgmental embedding of session types in rust. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, volume 222 of *LIPIcs*, pages 22:1–22:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ECOOP.2022.22.

26 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.

27 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`.

28 Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. A gentle introduction to multiparty asynchronous session types. In Marco Bernardo and Einar Broch Johnsen, editors, *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures*, volume 9104 of *Lecture Notes in Computer Science*, pages 146–178. Springer, 2015. `doi:10.1007/978-3-319-18941-3\_4`.

29 Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. *Theor. Comput. Sci.*, 802:38–66, 2020. `doi:10.1016/j.tcs.2019.07.005`.

30 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-free asynchronous message reordering in rust with multiparty session types. In Jaejin Lee, Kunal Agrawal, and Michael F. Spear, editors, *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, pages 246–261. ACM, 2022. `doi:10.1145/3503221.3508404`.

31 Francesco Dagnino, Paola Giannini, and Mariangiola Dezani-Ciancaglini. Deconfined global types for asynchronous sessions. In Ferruccio Damiani and Ornela Dardha, editors, *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*, volume 12717 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2021. `doi:10.1007/978-3-030-78142-2\_3`.

32 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`.

33 Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. Resource-aware session types for digital contracts. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*, pages 1–16. IEEE, 2021. `doi:10.1109/CSF51468.2021.00004`.

34 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`.

35 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`.

36 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`.

37 Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010. `doi:10.1017/S0956796809990268`.

38 Blaise Genest, Dietrich Kuske, and Anca Muscholl. On communicating automata with bounded channels. *Fundam. Inform.*, 80(1-3):147–167, 2007. URL: `http://content.iospress.com/articles/fundamenta-informaticae/fi80-1-3-09`.

39 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`.

40 Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi, and Pascal Weisenburger. Multiparty languages: The choreographic and multitier cases (pearl). In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 22:1–22:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ECOOP.2021.22`.

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

42 Cinzia Di Giusto, Laetitia Laversa, and Étienne Lozes. On the k-synchronizability of systems. In Jean Goubault-Larrecq and Barbara König, editors, *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 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 12077 of *Lecture Notes in Computer Science*, pages 157–176. Springer, 2020. `doi:10.1007/978-3-030-45231-5\_9`.

43 Paul Harvey, Simon Fowler, Ornela Dardha, and Simon J. Gay. Multiparty session types for safe runtime adaptation in an actor language. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 10:1–10:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ECOOP.2021.10`.

44 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`.

45 Andrew K. Hirsch and Deepak Garg. Pirouette: higher-order typed functional choreographies. *Proc. ACM Program. Lang.*, 6(POPL):1–27, 2022. `doi:10.1145/3498684`.

46 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`.

47 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`.

48 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`.

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

50 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.

51 Atsushi Igarashi, Peter Thiemann, Yuya Tsuda, Vasco T. Vasconcelos, and Philip Wadler. Gradual session types. *J. Funct. Program.*, 29:e17, 2019. doi:10.1017/S0956796819000169.

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

53 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proc. ACM Program. Lang.*, 6(ICFP):466–495, 2022. doi:10.1145/3547638.

54 Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for rust. In Patrick Bahr and Sebastian Erdweg, editors, *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2015, Vancouver, BC, Canada, August 30, 2015*, pages 13–22. ACM, 2015. doi:10.1145/2808098.2808100.

55 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.

56 Alex C. Keizer, Henning Basold, and Jorge A. Pérez. Session coalgebras: A coalgebraic view on regular and context-free session types. *ACM Trans. Program. Lang. Syst.*, 44(3):18:1–18:45, 2022. doi:10.1145/3527633.

57 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. URL: http://arxiv.org/abs/1902.01353, arXiv:1902.01353.

58 Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in go using behavioural types. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1137–1148. ACM, 2018. doi:10.1145/3180155.3180157.

59 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.

60 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.

61 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.

62 Sam Lindley and J. Garrett Morris. Embedding session types in haskell. In Geoffrey Mainland, editor, *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, pages 133–145. ACM, 2016. doi:10.1145/2976002.2976018.

63 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.

64 Rupak Majumdar, Madhavan Mukund, Felix Stutz, and Damien Zufferey. Generalising projection in asynchronous multiparty session types. In Serge Haddad and Daniele Varacca, editors, *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference*, volume 203 of *LIPIcs*, pages 35:1–35:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CONCUR.2021.35.

65 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.

66 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.

67 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.

68 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.

69 Anca Muscholl and Doron A. Peled. Message sequence graphs and decision problems on mazurkiewicz traces. In Miroslaw Kutylowski, Leszek Pacholski, and Tomasz Wierzbicki, editors, *Mathematical Foundations of Computer Science 1999, 24th International Symposium, MFCS'99, Szklarska Poreba, Poland, September 6-10, 1999, Proceedings*, volume 1672 of *Lecture Notes in Computer Science*, pages 81–91. Springer, 1999. doi:10.1007/3-540-48340-3_8.

70 Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: compile-time API generation of distributed protocols with refinements in f#. In Christophe Dubach and Jingling Xue, editors, *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*, pages 128–138. ACM, 2018. doi:10.1145/3178372.3179495.

71 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.

72 Kirstin Peters, Uwe Nestmann, and Christoph Wagner. Fault-tolerant multiparty session types. In Mohammad Reza Mousavi and Anna Philippou, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 42nd IFIP WG 6.1 International Conference, FORTE 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings*, volume 13273 of *Lecture Notes in Computer Science*, pages 93–113. Springer, 2022. doi:10.1007/978-3-031-08679-3_7.

73 Emil L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52:264–268, 1946.

74 Alceste Scalas and Nobuko Yoshida. Lightweight session programming in scala. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 21:1–21:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.21.

75 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.

76 Michael Sipser. *Introduction to the theory of computation.* PWS Publishing Company, 1997.

77 Felix Stutz and Damien Zufferey. Comparing channel restrictions of communicating state machines, high-level message sequence charts, and multiparty session types. In Pierre Ganty and Dario Della Monica, editors, *Proceedings of the 13th International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2022, Madrid, Spain, September 21-23, 2022*, volume 370 of *EPTCS*, pages 194–212, 2022. doi:10.4204/EPTCS.370.13.

78 Peter Thiemann. Intrinsically-typed mechanized semantics for session types. In Ekaterina Komendantskaya, editor, *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, pages 19:1–19:15. ACM, 2019. doi:10.1145/3354166.3354184.

79 Peter Thiemann and Vasco T. Vasconcelos. Context-free session types. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 462–475. ACM, 2016. doi:10.1145/2951913.2951926.

80 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.

81 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.

82 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.

83 Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. A multiparty session typing discipline for fault-tolerant event-driven distributed programming. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–30, 2021. doi:10.1145/3485501.

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

85 Nobuko Yoshida and Lorenzo Gheri. A very gentle introduction to multiparty session types. In Dang Van Hung and Meenakshi D'Souza, editors, *Distributed Computing and Internet Technology - 16th International Conference, ICDCIT 2020, Bhubaneswar, India, January 9-12, 2020, Proceedings*, volume 11969 of *Lecture Notes in Computer Science*, pages 73–93. Springer, 2020. doi:10.1007/978-3-030-36987-3_5.

86 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 Definitions for Section 2: Multiparty Session Types

A.1 Semantics of Communicating State Machines [77, App. A.4]

With $\mathsf{Chan} = \{\langle \mathtt{p}, \mathtt{q}\rangle \mid \mathtt{p}, \mathtt{q} \in \mathcal{P}, \mathtt{p} \neq \mathtt{q}\}$, we denote the set of channels. The set of global states of a CSM is given by $\prod_{\mathtt{p}\in\mathcal{P}} Q_{\mathtt{p}}$. Given a global state q, $q_{\mathtt{p}}$ denotes the state of $\mathtt{p}$ in q. A *configuration* of a CSM $\mathcal{A}$ is a pair (q, ξ), where q is a global state and $\xi : \mathsf{Chan} \to \mathcal{V}^\infty$ is a mapping of each channel to its current content. The initial configuration (q_0, ξ_ε) consists of a global state q_0 where the state of each role is the initial state $q_{0,\mathtt{p}}$ of $A_{\mathtt{p}}$ and a mapping ξ_ε, which maps each channel to the empty word ε. A configuration (q, ξ) is said to be *final* iff each individual local state $q_{\mathtt{p}}$ is final for every $\mathtt{p}$ and ξ is ξ_ε.

The global transition relation $\to$ 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 role $\mathtt{r} \neq \mathtt{p}$, $\xi'(\langle \mathtt{p}, \mathtt{q}\rangle) = \xi(\langle \mathtt{p}, \mathtt{q}\rangle) \cdot m$ 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 role $\mathtt{r} \neq \mathtt{q}$, $\xi(\langle \mathtt{p}, \mathtt{q}\rangle) = m \cdot \xi'(\langle \mathtt{p}, \mathtt{q}\rangle)$ and $\xi'(c) = \xi(c)$ for every other channel $c \in \mathsf{Chan}$.
- $(q, \xi) \xrightarrow{\varepsilon} (q', \xi)$ 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 always starts with an initial configuration (q_0, ξ_0), and is a finite or infinite sequence $(q_0, \xi_0) \xrightarrow{w_0} (q_1, \xi_1) \xrightarrow{w_1} \ldots$ for which $(q_i, \xi_i) \xrightarrow{w_i} (q_{i+1}, \xi_{i+1})$. The word $w_0 w_1 \ldots \in \Sigma^\infty$ is said to be the *trace* of the run. A run is called maximal if it is either infinite or finite and ends in a final configuration. As before, the trace of a maximal run is maximal. The language $\mathcal{L}(\mathcal{A})$ of the CSM $\mathcal{A}$ consists of its set of maximal traces.

B Additional Explanation for Different Merge Operators on FSMs from Section 3

Visual Explanation of the Parametric Projection Operator: Collapsing Erasure Here, we describe *collapsing erasure* more formally. Let $\mathbf{G}$ be some global type and $\mathtt{r}$ be the role on to which we project. We apply the parametric projection operator to the state machine $\mathsf{GAut}(\mathbf{G})$. It projects each transition label on to the respective event for role $\mathtt{r}$: every forward transition label $\mathtt{p}\to\mathtt{q}:m$ turns to $\mathtt{p}\triangleright\mathtt{q}!m$ if $\mathtt{r} = \mathtt{p}$, $\mathtt{p}\triangleleft\mathtt{q}?m$ if $\mathtt{r} = \mathtt{q}$, and ε otherwise. Then, it collapses neutral states with a single successor: $q_{1|2}$ replaces two states q_1 and q_2 if $q_1 \xrightarrow{\varepsilon} q_2$ is the only forward transition for q_1 and $q_1 \xrightarrow{x} q_2$ for $x \neq \varepsilon$ in $\delta_{\mathsf{GAut}(\mathbf{G})}$. In case there is only a backward transition from q_1 to q_2, the state $q_{1|2}$ is also final. This accounts for loops a role is not part of.

We call this procedure collapsing erasure as it erases interactions that do not belong to a role and collapses some states. It is common to all the presented merge operators. This procedure yields a state machine over $\Sigma_{\mathtt{r}}$. It is straightforward that it is still ancestor-recursive, free from intermediate recursion and non-merging. However, it might not be dense. In fact, it is not dense if $\mathtt{r}$ is not involved in some choice with more than one branch.

Parametric Merge in the Visual Explanation The parametric projection operator applies the merge operator for these cases. Visually, these correspond precisely to the remaining neutral states (since all neutral states with a single successor have been collapsed). For instance, we have a neutral state q_1 with $q_1 \xrightarrow{\varepsilon} q_2$ and $q_1 \xrightarrow{\varepsilon} q_3$ for $q_2 \neq q_3$. Through the parametric projection operator, the merge operator may be indirectly called recursively. Thus, we explain the merge operators for two states (and their cones) in general. No information is propagated when the merge operator recurses and recursion variables are never unfolded. Thus, we can ignore backward transitions and consider the cones of q_2 and q_3. Intuitively, we iteratively apply the merge operator from lower to higher levels. However, we might need descend again when merge is applied recursively. Similar to the syntactic version, we do only explain the 2-ary case but the reasoning easily lifts to the n-ary case.

Visual Explanation of Plain Merge The plain merge is not applied recursively. Thus, we consider q_1 with $q_1 \xrightarrow{\varepsilon} q_2$ and $q_1 \xrightarrow{\varepsilon} q_3$ for $q_2 \neq q_3$ such that q_1 has the lowest level for which this holds. Hence, we can assume that each cone of q_2 and q_3 does not contain neutral states. Then, the plain merge is only defined if there is an isomorphism between the states of both cones that satisfy the following conditions:

- it preserves the transition labels and hence the kind of states, and
- if a state has a backward transition to a state outside of the cone, its isomorphic state has a transition to the some state

If defined, the result is q_1 with its cone (and q_2 with its cone is removed).

Visual Explanation of Semi-full Merge The semi-full merge applies itself recursively. Thus, we consider two states $q_2 \neq q_3$ that shall be merged. As before, we can assume that each cone of q_2 and q_3 does not contain neutral states. In addition to plain merge, the semi-full merge allows to merge receive states. For these, we introduce a new receive state $q_{2|3}$ from which all new transitions start. For all possible transitions from q_2 and q_3, we check if there is a transition with the same label from the other state. For the ones not in common, we simply add the respective transition with the state it leads to and its respective cone. For the ones in common, we recursively check if the two states, which both transitions lead to, can be merged. If not, the semi-full merge is undefined. If so, we add the original transition to the state of the respective merge and keep its cone.

Visual Explanation of Full Merge Intuitively, the full merge simply applies the idea of the semi-full merge to another case. For the semi-full merge, one can recursively apply the merge operator when a reception was common between two states to merge. The full merge operator allows to descend for recursion variable binders.

C Formalisation for Section 4: Implementability for Global Types from MSTs is Decidable

C.1 Definitions for Section 4.1

▶ **Definition C.1** (Concatenation of MSCs [77]). *Let $M_i = (N_i, p_i, f_i, l_i, (\leq_p^i)_{\mathtt{p} \in \mathcal{P}})$ for $i \in \{1, 2\}$ where M_1 is a BMSC and M_2 is an MSC with disjoint sets of events, i.e., $N_1 \cap N_2 = \emptyset$. We define their* concatenation *$M_1 \cdot M_2$ as the MSC $M = (N, p, f, l, (\leq_p)_{\mathtt{p} \in \mathcal{P}})$ where:*

- $N \quad := \quad N_1 \, \cup \, N_2,$
- *for* $\zeta \in \{p, f, l\}: \quad \zeta(e) := \begin{cases} \zeta(e) & \text{if } e \in N_1 \\ \zeta(e) & \text{if } e \in N_2 \end{cases}$, *and*
- $\forall \mathtt{p} \in \mathcal{P}: \quad \leq_{\mathtt{p}} \quad := \quad \leq_{\mathtt{p}}^1 \, \cup \, \leq_{\mathtt{p}}^2 \, \cup \{(e_1, e_2) \mid e_1 \in N_1 \, \wedge \, e_2 \in N_2 \, \wedge p(e_1) = p(e_2) = \mathtt{p}\}.$

▶ **Definition C.2** (Language of an HMSC [77]). *Let $H = (V, E, v^I, V^T, \mu)$ be an HMSC. The language of H is defined as*

$$\mathcal{L}(H) := \{w \mid w \in \mathcal{L}(\mu(v_1)\mu(v_2)\dots\mu(v_n)) \text{ with } v_1 = v^I \wedge \forall\, 0 \leq i < n : (v_i, v_{i+1}) \in E \wedge v_n \in V^T\}$$
$$\cup \; \{w \mid w \in \mathcal{L}(\mu(v_1)\mu(v_2)\dots) \text{ with } v_1 = v^I \wedge \forall\, i \geq 0 : (v_i, v_{i+1}) \in E\} \, .$$

C.2 Proof of Lemma 4.8: Projection by Erasure is Correct

Let $H = (V, E, v^I, V^T, \mu)$ be an HMSC. For every $v \in V$, it is straightforward that the construction of $\mu(v)\Downarrow_{\mathtt{p}}$ yields $\mathcal{L}(\mu(v))\Downarrow_{\Sigma_{\mathtt{p}}} = \mathcal{L}(\mu(v)\Downarrow_{\mathtt{p}})$ (1). We recall that $\sim$ does not reorder events by the same role: $w \sim w'$ for $w \in \Sigma_{\mathtt{p}}$ iff $w = w'$ (2).

The following reasoning proves the claim where the first equivalence follows from the construction of the transition relation of $H\Downarrow_{\mathtt{p}}$:

$$\begin{aligned}
& w \in \mathcal{L}(H\Downarrow_{\mathtt{p}}) \\
\Leftrightarrow\quad & w = w_1 \ldots, \text{ there is a path } v_1, \ldots \text{ in } H \text{ and } w_i \in \mathcal{L}(\mu(v_i)\Downarrow_{\mathtt{p}}) \text{ for every } i \\
\overset{(1)}{\Leftrightarrow}\quad & w = w_1 \ldots, \text{ there is a path } v_1, \ldots \text{ in } H \text{ and } w_i \in \mathcal{L}(\mu(v_i))\Downarrow_{\Sigma_{\mathtt{p}}} \text{ for every } i \\
\overset{(2)}{\Leftrightarrow}\quad & w \in \mathcal{L}(H)\Downarrow_{\Sigma_{\mathtt{p}}}
\end{aligned}$$

C.3 Proof of Theorem 4.10: Erasure Candidate Implementation is Sufficient

We first use the correctness of the global type encoding (Theorem 4.5) to observe that $\mathcal{L}_{\text{fin}}(\mathbf{G}) = \mathcal{L}_{\text{fin}}(H(\mathbf{G}))$. Theorem 13 by Alur et al. [3] states that the canonical candidate implementation implements $\mathcal{L}_{\text{fin}}(H(\mathbf{G}))$ if it is implementable. Corollary 4.9 and the fact that the FSM for each role is deterministic by construction allows us to replace every $A_{\mathtt{p}}$ from the canonical candidate implementation with the projection by erasure $H(\mathbf{G})\Downarrow_{\mathtt{p}}$ for every role $\mathtt{p}$ which proves the claim. ◀

C.4 Proof of Lemma 4.11: Erasure Candidate Implementation Generalises to Infinite Case

Let us assume that $\mathbf{G}$ is implementable. From Theorem 4.10, we know that $\{\!\{H(\mathbf{G})\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ is deadlock free and $\mathcal{L}_{\text{fin}}(\{\!\{H(\mathbf{G})\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}) = \mathcal{L}_{\text{fin}}(\mathbf{G})$.We prove the claim by showing both inclusions.

First, we show that $\mathcal{L}_{\text{inf}}(\{\!\{H(\mathbf{G})\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}) \subseteq \mathcal{L}_{\text{inf}}(\mathbf{G})$. For this direction, let w be a word in $\mathcal{L}_{\text{inf}}(\{\!\{H(\mathbf{G})\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}})$. We need to show that there is a run ρ in $\mathsf{GAut}(\mathbf{G})$ such that $w \preceq^{\omega}_{\sim} \text{split}(\text{trace}(\rho))$. From the 0-Reachable-Assumption (p.15), we know that for every $u \in \text{pref}(w)$, it holds that $u \in \text{pref}(\mathcal{L}_{\text{fin}}(\mathbf{G}))$. Thus, there exists a finite run ρ (that does not necessarily end in a final state) and u' such that $u.u' \sim \text{trace}(\rho)$. We call ρ a witness run. Intuitively, we will need to argue that every such witness run for u can be extended when appending the next event x from w to obtain ux. In general, this does not hold for every choice of witness run. However, because of monotonicity, any run (or rather a prefix of it) for an extension ux can also be used as witness run for u. Thus, we make use of the idea of prophecy variables [1] and assume an oracle which picks the correct witness run for every prefix u. This oracle does not restrict the next possible events in any way. From here, we apply the same idea as Majumdar et al. for the proof of Lemma 41 [64]. We construct a tree $\mathcal{T}$ such that each node represents a run ρ of some finite prefix w' of w. The root's label is the empty run. For every node labelled with ρ, the children's extend ρ by a single transition. The tree $\mathcal{T}$ is finitely branching by construction of $\mathsf{GAut}(\mathbf{G})$ for every role $\mathtt{p}$. With König's Lemma, we obtain an infinite path in $\mathcal{T}$ and thus an infinite run ρ in $\mathsf{GAut}_{\text{async}}(\mathbf{G})$ with $w \preceq^{\omega}_{\sim} \text{trace}(\rho)$. From this, it follows that $w \in \mathcal{L}_{\text{inf}}(\mathbf{G})$.

Second, we show that $\mathcal{L}_{\text{inf}}(\mathbf{G}) \subseteq \mathcal{L}_{\text{inf}}(\{\!\{H(\mathbf{G})\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}})$. Let w be a word in $\mathcal{L}_{\text{inf}}(\mathbf{G})$. Eventually, we will apply the same reasoning with König's lemma to obtain an infinite run in $\{\!\{H(\mathbf{G})\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ for w. Inspired from the first statement of Lemma 25 by Majumdar et al. [64], we show:

(i) for every prefix $w' \in \mathrm{pref}(w)$, there is a run ρ' in $\{\!\{H(\mathbf{G})\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ such that $w' \preceq \mathrm{trace}(\rho')$, and

(ii) for every extension $w'x$ where x is the next event in w, the run ρ' can be extended

We prove Claim (i) first. We first observe that, with the 0-Reachable-Assumption (p.15), there is an extension w'' of w' with $w'' \in \mathcal{L}(\mathbf{G})$. By construction, we know that there is a run ρ'' in $\{\!\{H(\mathbf{G})\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ for w''. For ρ', we can simply take the prefix of ρ'' which matches w'. This proves Claim (i).

Now, let us prove Claim (ii). Similar to the first case, we will use prophecy variables and an oracle to pick the correct witness run that we can extend. Again, because of monotonicity, any run (or rather a prefix of it) for an extension $w'x$ can also be used as witness run for w'. As before, we make use of the idea of prophecy variables [1], assume an oracle which picks the correct witness run for every prefix w', and this oracle does not restrict the roles in any way. From this, Claim (ii) follows.

From here, we (again) use the same reasoning as Majumdar et al. for the proof of Lemma 41 [64]. We construct a tree $\mathcal{T}$ such that each node represents a run ρ of some finite prefix w' of w. The root's label is the empty run. For every node labelled with ρ, the children's extend ρ by a single transition. The tree $\mathcal{T}$ is finitely branching by construction of $A_{\mathtt{p}}$ for every role $\mathtt{p}$. With König's Lemma, we obtain an infinite path in $\mathcal{T}$ and, thus, an infinite run ρ in $\{\!\{H(\mathbf{G})\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ with $w \preceq^{\omega}_{\sim} \mathrm{trace}(\rho)$. From this, it follows that $w \in \mathcal{L}(\{\!\{H(\mathbf{G})\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}})$. ◀

C.5 Formalisation for Lemma 4.15: Implementability entails Globally Cooperative

▶ **Definition C.3** (Matching Sends and Receptions (Def. 2 [77])). *In a word $w = e_1 \ldots \in \Sigma^\infty$, a send event $e_i = \mathtt{p} \triangleright \mathtt{q}!m$ is* matched *by a receive event $e_j = \mathtt{q} \triangleleft \mathtt{p}?m$, denoted by $e_i \vdash\!\dashv e_j$, if $i < j$ and $\mathcal{V}((e_1 \ldots e_i)\Downarrow_{\mathtt{p}\triangleright\mathtt{q}!\_}) = \mathcal{V}((e_1 \ldots e_j)\Downarrow_{\mathtt{q}\triangleleft\mathtt{p}?\_})$. A send event e_i is* unmatched *if there is no such receive event e_j.*

Proof. We prove our claim by contraposition: assume there is a loop $v_1, \ldots, v_n$ such that the communication graph of $\mu(v_1) \ldots \mu(v_n)$ is not weakly connected. By construction of $H(\mathbf{G})$, we know that every vertex is reachable so there is a path $u_1 \ldots u_m v_1 \ldots v_n$ in $H(\mathbf{G})$ for some m and vertices u_1 to u_n such that $u_1 = v^I$. By the 0-Reachable-Assumption (p.15), this path can be completed to end in a terminal vertex to obtain $u_1 \ldots u_m v_1 \ldots v_n u_{m+1} \ldots u_{m+k}$ for some k and vertices u_{m+1} to u_{m+k} such that $u_{k+m} \in V^T$. By the syntax of global types and the construction of $H(\mathbf{G})$, there is a role $\mathtt{p}$ that is the (only) sender in v_1 and u_{m+1}.

Without loss of generality, let $\mathcal{S}_1$ and $\mathcal{S}_2$ be the two sets of (active) roles whose communication graphs of $v_1 \ldots v_n$ are weakly connected and their union consists of all active roles. Similar reasoning applies if there are more than two sets.

We want to consider the specific linearisations from the language of the BMSC of each subpath. Intuitively, these simply follow the order prescribed by the global type and do not exploit the partial order of BMSC or the closure of the semantics for global types. For this, we say that w_1 is the *canonical word for path* $u_1, \ldots u_m$ if $w_1 \in \{w'_1 \ldots w'_m \mid w'_i \in \mathcal{L}(\mu(u_i))$ for $1 \leq i \leq m\}$. Analogously, let w_2 be the canonical word for $v_1 \ldots v_n$ and w_3 be the canonical word for $u_{m+1} \ldots u_{m+k}$. Without loss of generality, $\mathcal{S}_1$ contains the sender of the first element in w_2 and w_3 — basically the role which decides when to exit the loop for the considered loop branch. Let $\{\!\{H(\mathbf{G})\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ be the erasure candidate implementation.

By its definition and the correctness of $H(\mathbf{G})$, it holds that: $\mathcal{L}(\mathbf{G}) = \mathcal{L}(H(\mathbf{G}))$. With the equivalence of the canonical candidate implementation (Corollary 4.9), the reasoning for Lemma 3.2 by Lohrey [63], and the fact that it generalises to infinite executions Lemma 4.11, the erasure candidate implementation admits at least the language specified by $H(\mathbf{G})$:

$\mathcal{L}(H(\mathbf{G})) \subseteq \mathcal{L}(\{\!\{H(\mathbf{G})\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}})$.

Thus, it holds that $\mathcal{L}(\mathbf{G}) = \mathcal{L}(\{\!\{H(\mathbf{G})\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}})$ if $\mathbf{G}$ is implementable. Therefore, we know that $w_1 . w_2 . w_3 \in \mathcal{L}(\{\!\{H(\mathbf{G})\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}})$.

From the construction of $H(\mathbf{G})$ and the construction of w_i for $i \in \{1, 2, 3\}$, it also holds that $w_1 . (w_2)^h . w_3 \in \mathcal{L}(H(\mathbf{G})) \subseteq \mathcal{L}(\{\!\{H(\mathbf{G})\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}})$ for any $h > 0$.

By construction of $\mathcal{S}_1$ and $\mathcal{S}_2$, no two roles from both sets communicate with each other in w_2: there are no $\mathtt{r} \in \mathcal{S}_1$ and $\mathtt{s} \in \mathcal{S}_2$ such that $\mathtt{r} \triangleright \mathtt{s}!m$ is in w_2 or $\mathtt{s} \triangleright \mathtt{r}!m$ is in w_2 (and consequently $\mathtt{r} \triangleleft \mathtt{s}?m$ is in w_2 or $\mathtt{s} \triangleleft \mathtt{r}?m$ is in w_2) for any m.

From the previous two observations, it follows that

$w_1 . w_2 . (w_2\Downarrow_{\Sigma_{\mathcal{S}_1}})^h . w_3 \in \mathcal{L}(\{\!\{H(\mathbf{G})\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}})$

for any h where $\Sigma_{\mathcal{S}_1} = \bigcup_{\mathtt{r}\in\mathcal{S}_1} \Sigma_{\mathtt{r}}$. Intuitively, this means that the set of roles with the role to decide when to exit the loop can continue longer in the loop than the roles in $\mathcal{S}_2$.

With $\mathcal{L}(\mathbf{G}) = \mathcal{L}(H(\mathbf{G}))$, it suffices to show the following to find a contradiction: $w_1 . w_2 . (w_2\Downarrow_{\Sigma_{\mathcal{S}_1}}) . w_3 \notin \mathcal{L}(H(\mathbf{G}))$.

Towards a contradiction, we assume the membership holds. By determinacy of $H(\mathbf{G})$, we need to find a path $v'_1 \ldots v'_{m'}$, that starts at the beginning of the loop, i.e., $v'_1 = v_1$, with canonical word w_4 such that $w_2\Downarrow_{\Sigma_{\mathcal{S}_1}} . w_3 \sim w_4$.

We show such a path cannot exist and that we would need to diverge during the loop.

For readability, we denote $w_2 . w_3$ with x and $w_2\Downarrow_{\Sigma_{\mathcal{S}_1}} . w_3$ with x'. We know that x' is a subsequence of x, i.e., $x' = x'_1 \ldots x'_l$ and $x = x_1 \ldots x_{l'}$. Let $x_1 \ldots x_j = x'_1 \ldots x'_j$ denote the maximal prefix on which both agree. Since $\mathcal{S}_2$ is not empty, we know that j can be at most $|w_2\Downarrow_{\Sigma_{\mathcal{S}_1}}|$. (Intuitively, j cannot be so big that it reaches w_3 because there will be mismatches due to $w_2\Downarrow_{\Sigma_{\mathcal{S}_2}}$ before.) We also claim that the next event x_{j+1} cannot be a receive event. If it was, there was a matching send event in $x_1 \ldots x_j$ (which is equal to $x'_1 \ldots x'_j$ by construction). Such a matching send event exists by construction of x from a path in $H(\mathbf{G})$. By definition of $\Downarrow_{\_}$, the matching receive event must be x'_{j+1} which would contradict the maximality of j. Thus, x_{j+1} must be a send event.

By determinacy of $H(\mathbf{G})$ and $j \leq |w_2\Downarrow_{\Sigma_{\mathcal{S}_1}}|$, we know that $x_1 \ldots x_j = x'_1 \ldots x'_j$ share a path $v_1 \ldots v_{n'}$ which is a part of the loop, i.e., $x_1 \ldots x_j \in \mathcal{L}(\mu(v_1) \cdots \mu(v_{n'}))$ with $n < n'$. For $M(\mathtt{p}\to\mathtt{q}\colon m)$ — the BMSC with solely this interaction from Definition 4.4, we say that $\mathtt{p}$ is its *sender*. The syntax of global types prescribes that choice is deterministic and the sender in a choice is unique. This is preserved for $H(\mathbf{G})$: for every vertex, all its successors have the same sender. Therefore, the path for x' can only diverge, but also needs to diverge, from the loop $v_1 \ldots v_n$ after the common prefix $v_1 \ldots v_{n'}$ with a different send event but with the same sender. Let v_l be next vertex after $v_1 \ldots v_{n'}$ on the loop $v_1, \ldots, v_n$ for which $\mu(v_l)$ is not M_ε — the BMSC with an empty set of event nodes from Definition 4.4. Note that x_{j+1} belongs to v_l: $x_{j+1} \in \mathrm{pref}(\mathcal{L}(\mu(v_l)))$.

We do another case analysis whether x_{j+1} belongs to $\mathcal{S}_1$ or not, i.e., if $x_{j+1} \in \Sigma_{\mathcal{S}_1}$.

If $x_{j+1} \notin \Sigma_{\mathcal{S}_1}$, there cannot be a path that continues for x'_{j+1} as the sender for $\mu(v_i)$ is not in $\mathcal{S}_1$. If $x_{j+1} \in \Sigma_{\mathcal{S}_1}$, the choice of j was not maximal which yields a contradiction. ◀

C.6 Further Explanation for Example 4.16

Here, we show that any trace of the CSM is specified by the HMSC. Let us consider a finite execution of the CSM for which we want to find a path in the HMSC. Let us assume there are i interactions between p and q and j interactions between r and s. In our asynchronous setting, these interactions are split and can be interleaved. From the CSM, it is easy to see that i is at least 2 and j is at least 1. The simplest path goes through the first loop once and accounts for $i-1$ iterations in the second loop and $j-1$ iterations in the third one. A more involved path could account for $\min(i,j)-1$ iterations of the first loop, as many as possible, and $i-\min(i,j)+1$ iterations of the second loop as well as $j-\min(i,j)+1$ iterations of the third loop. The key that both paths are valid possibilities is that the interactions of p and q in the first and second loop are indistinguishable, i.e., the executions can be reordered with $\sim$ such that both is possible. The syntactic restriction on choice does prevent this for global types (and this protocol cannot be represented with a global type). Intuitively, one cannot make up for a different number of loop iterations, that are the consequence of missing synchronisation, in global types because the "loop exit"-message will be distinct (compared to staying in the loop) and anything specified afterwards cannot be reordered by $\sim$ in front of it. It is straightforward to adapt the protocol so final states do not have outgoing transitions. We add another vertex with a BMSC at the bottom, which has the same structure as the top one but with another message l instead of m. We add an edge from the previous terminal vertex to the new vertex and make the new one the only terminal vertex. With this, p and r can eventually decide not to send m anymore and indicate their choice with the distinct message l to the other two roles.

D Proof for Lemma 5.4: Correctness of Algorithm 1 to check $\mathcal{I}$-closedness of Global Types

It is obvious that the language is preserved by the changes to the state machine. (We basically turned an unambiguous state machine into a deterministic one.)

For soundness, we assume that Algorithm 1 returns *true* and let w be a word in $\mathcal{C}^{\equiv_\mathcal{I}}(\mathcal{L}(\mathsf{GAut}(\mathbf{G})))$. By definition, there is a run with trace w' in $\mathsf{GAut}(\mathbf{G})$ such that $w' \equiv_\mathcal{I} w$. The conditions in Algorithm 1 ensure that $w = w'$ because no two adjacent elements in w' can be reordered with $\equiv_\mathcal{I}$. Therefore, $w \in \mathcal{L}(\mathsf{GAut}(\mathbf{G}))$ which proves the claim.

For completeness, we assume that the algorithm returns *false* and show that there is $w \in \mathcal{C}^{\equiv_\mathcal{I}}(\mathcal{L}_{\text{fin}}(\mathbf{G}))$ such that $w \notin \mathcal{L}_{\text{fin}}(\mathsf{GAut}(\mathbf{G}))$. Without loss of generality, let q_2 be the state for which an incoming label x and outgoing label y can be reorderd, i.e., $x \equiv_\mathcal{I} y$, and let q_1 be the state from which the transition with label x originates: $q_1 \xrightarrow{x} q_2 \in \delta_{\mathsf{GAut}(\mathbf{G})}$. We consider a word w' which is the trace of a maximal run that passes q and the transitions labelled with x and y. By construction, it holds that $w' \in \mathcal{L}_{\text{fin}}(\mathsf{GAut}(\mathbf{G}))$. We swap x and y in w' to obtain w. We denote x with $\mathtt{p}\to\mathtt{q}:m$ and y with $\mathtt{r}\to\mathtt{s}:m'$ such that $\{\mathtt{p},\mathtt{q}\}\cap\{\mathtt{r},\mathtt{s}\} \neq \emptyset$. From the syntactic restrictions of global types, we know that any transition label from q_1 has sender p while every transition label from q_2 has sender r. Because of this and determinacy of the state machine, there is no run in $\mathsf{GAut}(\mathbf{G})$ with trace w'. Thus, $w \notin \mathcal{L}_{\text{fin}}(\mathsf{GAut}(\mathbf{G}))$ which concludes the proof. ◀

E Proof for Theorem 6.7: Implementability with regard to Intra-role Reordering for Global Types from MSTs is Undecidable

Let $\{(u_1, u_2, \ldots, u_n), (v_1, v_2, \ldots, v_n)\}$ be an instance of MPCP where 1 is the special index with which each solution needs to start with. We construct a global type where, for a word $w = a_1 a_2 \cdots a_m \in \Delta^*$, a message labelled $[w]$ denotes a sequence of individual message interactions with message a_1, a_2, …, a_m, each of size 1. We define a parametric global type

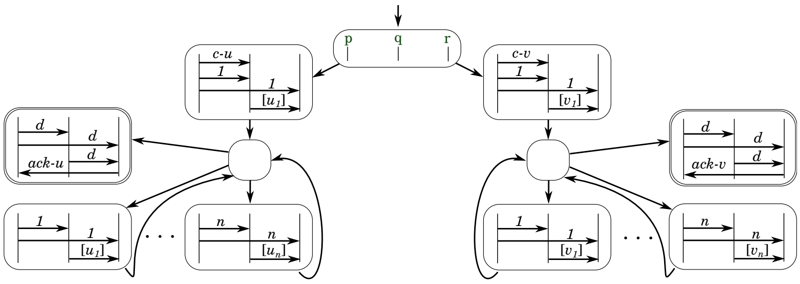


Figure 8 HMSC encoding $H(\mathbf{G}_{\mathrm{MPCP}})$ of the MPCP encoding (same as in main text)

where $x \in \{u, v\}$:

$$G(x, X) \quad := \quad \mathtt{p}\to\mathtt{q}: c\text{-}x.\, \mathtt{p}\to\mathtt{q}: 1.\, \mathtt{p}\to\mathtt{r}: 1.\, \mathtt{q}\to\mathtt{r}: [x_1].\, \mu t_1.\ + \begin{cases} \mathtt{p}\to\mathtt{q}: 1.\, \mathtt{p}\to\mathtt{r}: 1.\, \mathtt{q}\to\mathtt{r}: [x_1].\, t_1 \\ \dots \\ \mathtt{p}\to\mathtt{q}: n.\, \mathtt{p}\to\mathtt{r}: n.\, \mathtt{q}\to\mathtt{r}: [x_n].\, t_1 \\ \mathtt{p}\to\mathtt{q}: d.\, \mathtt{p}\to\mathtt{r}: d.\, \mathtt{q}\to\mathtt{r}: d.\, X \end{cases}$$

where $c\text{-}x$ indicates *choosing* tile set x. Using this, we obtain our encoding:

$$\mathbf{G}_{\mathrm{MPCP}} \quad := \quad + \begin{cases} G(u, \mathtt{r}\to\mathtt{p}: c\text{-}u.\, 0) \\ G(v, \mathtt{r}\to\mathtt{p}: c\text{-}v.\, 0) \end{cases} .$$

Figure 8 illustrates its HMSC encoding $H(\mathbf{G}_{\mathrm{MPCP}})$.

It suffices to show the following equivalences:

$$\begin{aligned} & \mathbf{G}_{\mathrm{MPCP}} \text{ is } \approx\text{-implementable} \\ \Leftrightarrow_1 \quad & \mathcal{C}^{\approx}(\mathcal{L}(G(u, 0)))\Downarrow_{\Sigma_{\mathtt{r}}} \cap\, \mathcal{C}^{\approx}(\mathcal{L}(G(v, 0)))\Downarrow_{\Sigma_{\mathtt{r}}} = \emptyset \\ \Leftrightarrow_2 \quad & \text{MPCP instance has no solution} \end{aligned}$$

We prove $\Rightarrow_1$ by contraposition. Let $w \in \mathcal{C}^{\approx}(\mathcal{L}(G(u, 0)))\Downarrow_{\Sigma_{\mathtt{r}}} \cap\, \mathcal{C}^{\approx}(\mathcal{L}(G(v, 0)))\Downarrow_{\Sigma_{\mathtt{r}}}$. For $x \in \{u, v\}$, let $w_x \in \mathcal{C}^{\approx}(\mathcal{L}(G(x, 0)))$ such that $w_x\Downarrow_{\Sigma_{\mathtt{r}}} = w$. By construction of $\mathbf{G}_{\mathrm{MPCP}}$, we know that $w_x \,.\, \mathtt{r} \triangleright \mathtt{p}!\mathit{ack}\text{-}x \,.\, \mathtt{p} \triangleleft \mathtt{r}?\mathit{ack}\text{-}x \in \mathcal{C}^{\approx}(\mathcal{L}(\mathbf{G}_{\mathrm{MPCP}}))$.

Suppose that CSM $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ $\approx$-implements $\mathbf{G}_{\mathrm{MPCP}}$. Then, it holds that

$$w_x \,.\, \mathtt{r} \triangleright \mathtt{p}!\mathit{ack}\text{-}x \,.\, \mathtt{p} \triangleleft \mathtt{r}?\mathit{ack}\text{-}x \in \mathcal{C}^{\approx}(\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}))$$

by (ii) from Definition 6.3. We also know that $w_x \,.\, \mathtt{r} \triangleright \mathtt{p}!\mathit{ack}\text{-}y \,.\, \mathtt{p} \triangleleft \mathtt{r}?\mathit{ack}\text{-}y \notin \mathcal{C}^{\approx}(\mathcal{L}(\mathbf{G}_{\mathrm{MPCP}}))$ for $x \neq y$ where $x, y \in \{u, v\}$. By the choice of w_u and w_v, it holds that $w_u\Downarrow_{\Sigma_{\mathtt{r}}} = w = w_v\Downarrow_{\Sigma_{\mathtt{r}}}$. Therefore, $\mathtt{r}$ needs to be in the same state of $A_{\mathtt{r}}$ after processing $w_u\Downarrow_{\Sigma_{\mathtt{r}}}$ or $w_v\Downarrow_{\Sigma_{\mathtt{r}}}$ and it can either send both $\mathit{ack}\text{-}u$ and $\mathit{ack}\text{-}v$, only one of them or none of them to $\mathtt{p}$. Thus, either one of the following is true:

a) (sending both) $w_x \,.\, \mathtt{r} \triangleright \mathtt{p}!\mathit{ack}\text{-}y \in \mathrm{pref}(\mathcal{C}^{\approx}(\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}})))$ for $x \neq y$ where $x, y \in \{u, v\}$, or
b) (sending u without loss of generality) $w_v \,.\, \mathtt{r} \triangleright \mathtt{p}!\mathit{ack}\text{-}u \notin \mathrm{pref}(\mathcal{C}^{\approx}(\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}})))$, or
c) (sending none) $w_x \,.\, \mathtt{r} \triangleright \mathtt{p}!\mathit{ack}\text{-}x \notin \mathrm{pref}(\mathcal{C}^{\approx}(\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}})))$ for $x \in \{u, v\}$.

All cases lead to deadlocks in $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$. For a) and for b) if $c\text{-}v$ was chosen in the beginning, $\mathtt{p}$ cannot receive the sent message as it disagrees with its choice from the beginning $c\text{-}x$. In all other cases, $\mathtt{p}$ waits for a message while no message will ever be sent. Having deadlocks contradicts the assumption that $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ $\approx$-implements $\mathbf{G}$ (and there cannot be any CSM that $\approx$-implements $\mathbf{G}$).

We prove $\Leftarrow_1$ next. The language $\mathcal{C}^{\approx}(\mathcal{L}(\mathbf{G}_{\mathrm{MPCP}}))$ is obviously non-empty. Therefore, let $w' \in \mathcal{C}^{\approx}(\mathcal{L}(\mathbf{G}_{\mathrm{MPCP}}))$. We split w to obtain:

$$w' = w \,.\, \mathtt{r} \triangleright \mathtt{p}!\mathit{ack}\text{-}x \,.\, \mathtt{p} \triangleleft \mathtt{r}?\mathit{ack}\text{-}x \text{ for some } w \text{ and } x \in \{u, v\}.$$

By construction of $\mathbf{G}_{\mathrm{MPCP}}$, we know that

$$w \in \mathcal{C}^{\approx}(\mathcal{L}(G(u, 0))) \;\cup\; \mathcal{C}^{\approx}(\mathcal{L}(G(v, 0))).$$

By assumption, it follows that exactly one of the following holds:

$$w\Downarrow_{\Sigma_{\mathtt{r}}} \in \mathcal{C}^{\approx}(\mathcal{L}(G(u,0)))\Downarrow_{\Sigma_{\mathtt{r}}} \quad \text{or} \quad w\Downarrow_{\Sigma_{\mathtt{r}}} \in \mathcal{C}^{\approx}(\mathcal{L}(G(v,0)))\Downarrow_{\Sigma_{\mathtt{r}}}.$$

We give a $\approx$-implementation for $\mathbf{G}_{\mathrm{MPCP}}$. It is straightforward to construct FSMs for both $\mathtt{p}$ and $\mathtt{q}$. They are involved in the initial decision and $\approx$ does not affect their projected languages. Thus, the projection by erasure can be applied to obtain FSMs $A_{\mathtt{p}}$ and $A_{\mathtt{q}}$. We construct an FSM $A_{\mathtt{r}}$ for $\mathtt{r}$ with control state $\mathtt{i} \in \{1,\ldots,n\}$, $\mathtt{j} \in \{1,\ldots,\max(|u_i| \mid i \in \{1,\ldots,n\})\}$, $\mathtt{d} \in \{0,1,2\}$, and $\mathtt{x} \in \{u,v\}$, where $|w|$ denotes the length of a word. The FSM is constructed in a way such that

$$w\Downarrow_{\Sigma_{\mathtt{r}}} \in \mathcal{C}^{\approx}(\mathcal{L}(G(u,0)))\Downarrow_{\Sigma_{\mathtt{r}}} \quad \text{if and only if} \quad \mathtt{d} \text{ is } 2 \text{ and } \mathtt{x} \text{ is } u \qquad \text{as well as}$$

$$w\Downarrow_{\Sigma_{\mathtt{r}}} \in \mathcal{C}^{\approx}(\mathcal{L}(G(v,0)))\Downarrow_{\Sigma_{\mathtt{r}}} \quad \text{if and only if} \quad \mathtt{d} \text{ is } 2 \text{ and } \mathtt{x} \text{ is } v.$$

We first explain that this characterisation suffices to show that $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ $\approx$-implements $\mathbf{G}$. The control state $\mathtt{d}$ counts the number of received d-messages. Thus, there will be no more messages to $\mathtt{r}$ in any channel once $\mathtt{d}$ is 2 by construction of $\mathbf{G}_{\mathrm{MPCP}}$. Once in a state for which $\mathtt{d}$ is 2, $\mathtt{r}$ sends message *ack-u* to $\mathtt{p}$ if $\mathtt{x}$ is u and message *ack-v* if $\mathtt{x}$ is v. With the characterisation, this message *ack-x* matches the message *c-x* sent from $\mathtt{p}$ to $\mathtt{q}$ in the beginning and, thus, $\mathtt{p}$ will be able to receive it and conclude the execution.

Now, we will explain how to construct the FSM $A_{\mathtt{r}}$. Intuitively, $\mathtt{r}$ keeps a tile number, which it tries to match against, and stores this in $\mathtt{i}$. It is initially set to 0 to indicate no tile has been chosen yet. The index $\mathtt{j}$ denotes the position of the letter it needs to match in tile $u_{\mathtt{i}}$ next and, thus, is initialised to 1. The variable $\mathtt{d}$ indicates the number of d-messages received so far, so initially $\mathtt{d}$ is 0. With this, $\mathtt{r}$ knows when it needs to send *ack-x*. The FSM for $\mathtt{r}$ tries to match the received messages against the tiles of u, so $\mathtt{x}$ is initialised to u. If this matching fails at some point, $\mathtt{x}$ is set to v as it learned that v was chosen initially by $\mathtt{p}$. In any of the following cases: if a received message is a d-message, $\mathtt{d}$ is solely increased by 1:

- If $\mathtt{x}$ is u and $\mathtt{i}$ is 0, $\mathtt{r}$ receives a message z from $\mathtt{p}$ and sets $\mathtt{i}$ to z (technically the integer represented by z).
- If $\mathtt{x}$ is u and $\mathtt{i}$ is not 0, $\mathtt{r}$ receives a message z from $\mathtt{q}$.
  - If z is the same as $u_{\mathtt{i}}[\mathtt{j}]$, we increment $\mathtt{j}$ by 1 and check if $\mathtt{j} > |u_{\mathtt{i}}|$ and, if so, set $\mathtt{i}$ to 0 and $\mathtt{j}$ to 1
  - If not, we set $\mathtt{x}$ to v
- Once $\mathtt{x}$ is v, $\mathtt{r}$ can simply receive all remaining messages in any order.

The described FSM can be used for $\mathtt{r}$ because it reliably checks whether a presented sequence of indices and words belongs to tile set u or v. It can do so because $\mathcal{C}^{\approx}(\mathcal{L}(G(u,0)))\Downarrow_{\Sigma_{\mathtt{r}}} \cap \mathcal{C}^{\approx}(\mathcal{L}(G(v,0)))\Downarrow_{\Sigma_{\mathtt{r}}} = \emptyset$ by assumption.

We prove $\Rightarrow_2$ by contraposition. Suppose the MPCP instance has a solution. Let i_1, …, i_k be a non-empty sequence of indices such that $u_{i_1}u_{i_2}\cdots u_{i_k} = v_{i_1}v_{i_2}\cdots v_{i_k}$ and $i_1 = 1$. It is easy to see that

$$w_x := \mathtt{r}\triangleleft\mathtt{p}?i_1\mathtt{r}\triangleleft\mathtt{q}?[x_{i_1}].\cdots.\mathtt{r}\triangleleft\mathtt{p}?i_k.\mathtt{r}\triangleleft\mathtt{q}?[x_{i_k}].\mathtt{r}\triangleleft\mathtt{p}?d.\mathtt{r}\triangleleft\mathtt{q}?d \in \mathcal{L}(G(x,0))\Downarrow_{\Sigma_{\mathtt{r}}} \text{ for } x \in \{u,v\}.$$

By definition of $\approx$, we can re-arrange the previous sequences such that

$$\mathtt{r}\triangleleft\mathtt{p}?i_1.\cdots.\mathtt{r}\triangleleft\mathtt{p}?i_k.\mathtt{r}\triangleleft\mathtt{q}?[x_{i_1}].\cdots.\mathtt{r}\triangleleft\mathtt{q}?[x_{i_k}].\mathtt{r}\triangleleft\mathtt{p}?d.\mathtt{r}\triangleleft\mathtt{q}?d \in \mathcal{C}^{\approx}(\mathcal{L}(G(x,0)))\Downarrow_{\Sigma_{\mathtt{r}}} \text{ for } x \in \{u,v\}.$$

Because $i_1,\ldots,i_k$ is a solution to the instance of MPCP, it holds that

$$\mathtt{r}\triangleleft\mathtt{q}?[u_{i_1}].\cdots.\mathtt{r}\triangleleft\mathtt{q}?[u_{i_k}] = \mathtt{r}\triangleleft\mathtt{q}?[v_{i_1}].\cdots.\mathtt{r}\triangleleft\mathtt{q}?[v_{i_k}]$$

and, thus,

$$\mathtt{r}\triangleleft\mathtt{p}?i_1.\cdots.\mathtt{r}\triangleleft\mathtt{p}?i_k.\mathtt{r}\triangleleft\mathtt{q}?[u_{i_1}].\cdots.\mathtt{r}\triangleleft\mathtt{q}?[u_{i_k}].\mathtt{r}\triangleleft\mathtt{p}?d.\mathtt{r}\triangleleft\mathtt{q}?d \text{ is in } \mathcal{C}^{\approx}(\mathcal{L}(G(v,0)))\Downarrow_{\Sigma_{\mathtt{r}}}.$$

This shows that $\mathcal{C}^{\approx}(\mathcal{L}(G(u,0)))\Downarrow_{\Sigma_{\mathtt{r}}} \cap \mathcal{C}^{\approx}(\mathcal{L}(G(v,0)))\Downarrow_{\Sigma_{\mathtt{r}}} \neq \emptyset$.

Lastly, we prove $\Leftarrow_2$. We know that the MPCP instance has no solution. Thus, there cannot be a non-empty sequence of indices $i_1, i_2, \ldots, i_k$ such that $u_{i_1} u_{i_2} \cdots u_{i_k} = v_{i_1} v_{i_2} \cdots v_{i_k}$ and $i_1 = 1$. For any possible word $w_u \in \mathcal{C}^{\approx}(\mathcal{L}(G(u, 0)))\Downarrow_{\Sigma_{\mathtt{r}}}$ and word $w_v \in \mathcal{C}^{\approx}(\mathcal{L}(G(v, 0)))\Downarrow_{\Sigma_{\mathtt{r}}}$.

We consider the sequence of receive events $w_x\Downarrow_{\mathtt{r}\triangleleft\mathtt{p}?\_}$ with sender $\mathtt{p}$ and the sequence of messages $w_x\Downarrow_{\mathtt{r}\triangleleft\mathtt{q}?\_}$ from $\mathtt{q}$ for $x \in \{u, v\}$. The intra-role indistinguishability relation $\approx$ allows to reorder events of both but for a non-empty intersection of both sets, we would still need to find a word w_u and w_v such that

$$w_u\Downarrow_{\mathtt{r}\triangleleft\mathtt{p}?\_} = w_v\Downarrow_{\mathtt{r}\triangleleft\mathtt{p}?\_} \quad \text{and} \quad w_u\Downarrow_{\mathtt{r}\triangleleft\mathtt{q}?\_} = w_v\Downarrow_{\mathtt{r}\triangleleft\mathtt{q}?\_}.$$

However, $G(x, 0)$ for $x \in \{u, v\}$ is constructed in a way that this is only possible if the MPCP instance has a solution. Therefore, the intersection is empty which proves our claim. ◀