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 sender-driven choice, which allow a sender to send to different receivers upon branching and a receiver to receive from different senders. 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 that 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.ECOOP.2023.32


Funding This research was sponsored in part by the Deutsche Forschungsgemeinschaft project 389792660 TRR 248 – CPEC.

Acknowledgements The author thanks Damien Zufferey, Emanuele D’Osualdo, Ashwani Anand, Rupak Majumdar, and the anonymous reviewers for their valuable feedback.

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 verifying such systems is algorithmically undecidable in general due to the combination of asynchrony (messages are buffered) and concurrency [13].

Multiparty Session Type (MST) frameworks [37, 38] provide efficient means to specify and verify such distributed message-passing systems (e.g., see the survey [5]). They have also been applied to various other domains like cyber-physical systems [46], timed systems [9], web services [65], and smart contracts [27]. 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., [54, 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.
Asynchronous MST Implementability is Decidable – Lessons Learned from MSCs

A global type provides a global view of the intended protocol. However, when implementing 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.

In a protocol in a distributed setting, one needs a local specification for each role. The first term implementations so the difference is not essential and we use the term role instead of process.

In general, one distinguishes between a role in a protocol and the process which implements its corresponding receive event. This protocol is specified with the following global type:

\[
G_{2BP} := \mu t. \{ a \to s: \text{query}. a \to a: \text{price}. + \{ a \to b: \text{split}. (b \to a: \text{yes}. a \to a: \text{buy}. t + b \to a: \text{no}. a \to s: \text{no}. t \}
\}
\]

The first term \( \mu t \) 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 by buyer \( a \) as it is the sender for the next interaction, e.g., \( a \to s: \text{query} \). For our asynchronous setting, this term jointly specifies the send event \( a \to s: \text{query} \) for buyer \( a \) and its corresponding receive event \( s \to a? \text{query} \) for seller \( s \), which may happen with arbitrary delay. The state machine in Figure 1a illustrates its semantics with abbreviated message labels.

**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 contrast, in a synchronous setting, there are no channels, yielding finite state systems. Still, we could not find a reference that precisely settles the decidability of synchronous implementability. We sketch a proof in Section 7. In this work, we solely deal with the asynchronous setting.

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 G_{2BP} while Figure 1b gives the projection onto seller s before the merge operator is applied – in both, messages are abbreviated with their first letter. It is easy to see that this procedure 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 or 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 sender-driven choice which is more permissive and allows these patterns. Among the classical projection operators, the one by Majumdar et al. [45] is the only to handle global types with sender-driven choice but it 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 projection/merge operators fail to project implementable variations of the two buyer protocol, showcasing the sources of incompleteness for the classical approach. For non-classical approaches, we refer to Section 7.

Lessons Learned from Message Sequence Charts

The two buyer protocol G_{2BP} can also be specified as high-level message sequence chart (HMSC) [47], as 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 its 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., [15, Fig. 1] and [38, Figs. 1 and 2]. The first formal connection was recently established by Stutz and Zufferey [57].
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 [2], 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 this result to infinite executions.\footnote{For this, we impose a mild assumption: all protocols can (but do not need to) terminate.}

Hence, algorithms and conditions center around checking implementability of HMSCs. In general, this problem is undecidable [44]. For \textit{globally-cooperative} HMSCs [31], Lohrey [44] proved it to be \textit{EXPSPACE}-complete. We show that any implementable global type belongs to this class of HMSCs.\footnote{For this, we impose a mild assumption: all protocols can (but do not need to) terminate.} These results give rise to the 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 [44] 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 \textit{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 this generalisation to be undecidable in general.

\textbf{Contributions and Outline}

We introduce our MST framework in Section 2 while Section 7 covers 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 sender-driven 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 $\Sigma$ be an alphabet. We denote the set of finite words over $\Sigma$ by $\Sigma^*$ and the set of infinite words by $\Sigma^{\omega}$. Their union is denoted by $\Sigma^\infty$. For two strings $u \in \Sigma^*$ and $v \in \Sigma^{\omega}$, 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 $u \preceq v$ while $\text{pref}(v)$ denotes all prefixes of $v$ and is lifted to languages as expected. 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 $p, q, r, s$. With $\Sigma_{\text{sync}} = \{ p \rightarrow q : m \mid p, q \in \mathcal{P} \text{ and } m \in \mathcal{V} \}$, 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: $\Sigma_p = \{ p \triangleright q!m, p \triangleleft q?m \mid q \in \mathcal{P}, m \in \mathcal{V} \}$ for a role $p$. For both send events $p \triangleright q!m$ and receive events $p \triangleleft q?m$, 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: $\Sigma = \bigcup_{p \in \mathcal{P}} \Sigma_p$. For the rest of this work, we fix the set of roles $\mathcal{P}$, the messages $\mathcal{V}$, and both sets $\Sigma_{\text{sync}}$ and $\Sigma$. We may also use the term $\Sigma_{\text{async}}$ for $\Sigma$. We define an operator that splits events from $\Sigma_{\text{sync}}$, $\text{split}(p \rightarrow q : m) := p \triangleright q!m, q \triangleleft p?m$, 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, $w|_{\mathcal{V} \triangleright p \triangleleft q}$ is the subsequence of $w$ with all of its send events where $p$ sends any message to $q$. 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. [45]. 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:

$$G ::= 0 \mid \sum_{i \in I} p \rightarrow q_i : m_i, G_i \mid \mu t. G \mid t$$

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

Our global types admit sender-driven choice as $p$ can send to different receivers upon branching: $\sum_{i \in I} p \rightarrow q_i : m_i, G_i$. This is also called generalised choice by Majumdar et al. [45]. In contrast, directed choice requires a sender to send to a single receiver, i.e., $\forall i, j \in I. q_i = q_j$. 


Asynchronous MST Implementability is Decidable – Lessons Learned from MSCs

Example 2.2 (Global types). The two buyer protocol $G_{2BP}$ from the introduction is a global type. Instead of $\sum$, we use $+$ with curly brackets.

Definition 2.3 (Syntax of local types). For a role $p$, the local types are defined as follows:

$$L ::= 0 \mid \oplus_{i \in I} q_i ! m_i, L \mid \&_{i \in I} q_i ? m_i, L \mid \mu t . L \mid t$$

We call $\oplus_{i \in I} q_i ! m_i$ an internal choice while $\&_{i \in I} q_i ? m_i$ is an external choice. For both, we require the choice to be unique, i.e., $i, j \in I. i \neq j \Rightarrow q_i \neq q_j \lor m_i \neq m_j$. 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 $G_{2BP}$, a local type for seller $s$ is

$$\mu t. \{ \begin{array}{ll} a ? \text{query} . a \text{price} . & (a ? \text{buy} . t \& a ? \text{no} . t) \end{array}$$

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). A state machine $A = (Q, \Delta, \delta, q_0, F)$ is a 5-tuple with a finite set of states $Q$, an alphabet $\Delta$, 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 $F$ for $Q \subseteq Q$. 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 $L(A)$ 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). We call $\{A_p\}_{p \in P}$ a communicating state machine (CSM) over $P$ and $V$ if $A_p$ is a finite state machine with alphabet $\Sigma_p$ for every $p \in P$. The state machine for $p$ is denoted by $(Q_p, \Sigma_p, \delta_p, q_{0p}, F_p)$. Intuitively, a CSM allows a set of state machines, one for each role in $P$, to communicate by sending and receiving messages. For this, each pair of roles $p, q \in P$, $p \neq q$, is connected by two directed message channels. A transition $q_p \xrightarrow{p @ m} q'_p$ in the state machine of $p$ denotes that $p$ sends message $m$ to $q$ if $p$ is in the state $q_p$ and changes its local state to $q'_p$. The channel $(p, q)$ is appended by message $m$. For receptions, a transition $q_q \xrightarrow{q @ m ?} q'_q$ in the state machine of $q$ corresponds to $q$ retrieving the message $m$ from the head of the channel when its local state is $q_q$ which is updated to $q'_q$. The run of a CSM always starts with empty channels and each finite state machine is in its respective initial state. A deadlock of $\{A_p\}_{p \in P}$ is a reachable configuration without outgoing transitions such that there is a non-empty channel or some participant is in a non-final local state. The formalisation of this intuition is standard and can be found in the technical report [56].

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: $a \rightarrow b : \text{cancel}$. $a \rightarrow s : \text{no}$. $0$. This is one of its traces: $a \triangleright b ! \text{cancel}$. $b \triangleleft a ? \text{cancel}$. $a \triangleright a ! \text{no}$. $s \triangleleft a ? \text{no}$. Because the active roles in
b ≪ a?cancel and a ≳ a!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: a ≳ b!cancel. a ≳ a!no. b ≪ a?cancel. a ≳ a!no.

Majumdar et al. [45] introduced the following relation to capture this phenomenon.

\textbf{Definition 2.8 (Indistinguishability relation \( \sim \) [45]).} We define a family of indistinguishability relations \( \sim_i \subseteq \Sigma^* \times \Sigma^* \), for \( i \geq 0 \). For \( w \in \Sigma^* \), we have \( w \sim_0 w \). For \( i = 1 \), we define:

1. If \( p \neq r \), then \( w.p \rightarrow q!m.r \rightarrow s!m'.u \sim_1 w.r \rightarrow s!m'.p \rightarrow q!m.u \).
2. If \( q \neq s \), then \( w.q \prec p?m.s \prec r?m'.u \sim_1 w.s \prec r?m'.q \prec p?m.u \).
3. If \( p \neq s \land (p \neq r \lor q \neq s) \), then \( w.p \rightarrow q!m.s \prec r?m'.u \sim_1 w.s \prec r?m'.p \rightarrow q!m.u \).
4. If \( |w|_{pq} \neq |w|_{qp} \), then \( w.p \rightarrow q!m.q \prec p?m'.u \sim_1 w.q \prec p?m'.p \rightarrow q!m.u \).

Let \( w, w' \), and \( w'' \) be words s.t. \( w \sim_1 w' \), and \( w' \sim_1 w'' \) 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 \( u \preceq \sim v \) if for each finite prefix \( w \) of \( u \), there is a finite prefix \( v' \) of \( v \) such that \( w \preceq \sim v' \). Define \( u \sim v \) iff \( u \preceq \sim v \) and \( v \preceq \sim u \).

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

For a language \( L \), we define \( C^\sim(L) = \{ w' \mid \forall w' \in \Sigma^* \exists w \in L \text{ and } w' \sim w \} \).

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

\textbf{Lemma 2.9 (L. 21 [45]).} Let \( \prod A_p \in P \) be a CSM. Then, \( L(\prod A_p) = C^\sim(L(\prod A_p)) \).

\section*{Global and Local Types – Semantics}

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

\textbf{Definition 2.10 (Semantics of global types).} We construct a state machine \( G\text{Aut}(G) \) to obtain the semantics of a global type \( G \). We index every syntactic subterm of \( G \) 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 \( G \) is \( [G, 1] \). For clarity, we do not quantify indices. We define \( G\text{Aut}(G) = (Q\text{Aut}(G), \Sigma_{\text{sync}}, \delta\text{Aut}(G), \lambda_0, G\text{Aut}(G), F_{\text{Aut}}(G)) \) where:

\( Q\text{Aut}(G) = \) the set of all indexed syntactic subterms \([G, k]\) of \( G \)
\( \delta\text{Aut}(G) = \) the smallest set containing \( ([\sum_{i \in I} p \rightarrow q_i; m_i; [G_i, k_i], k], p \rightarrow q; m, [G, k]) \)
\( \lambda_0, G\text{Aut}(G) = [G, 1] \), and \( F_{\text{Aut}}(G) = \{ [0, k] \mid k \text{ is an index for subterm } 0 \} \).

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 \( G \):

\[ L(G) := C^\sim(\text{split}(L(G\text{Aut}(G)))) \]
Asynchronous MST Implementability is Decidable – Lessons Learned from MSCs

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 $\varepsilon$ for transitions from recursion to binder states.

**Local Types.** For the semantics of local types, we analogously construct a state machine $\text{LAut}(\cdot)$. In contrast, we omit the closure $C\sim(\cdot)$ because languages of roles are closed under $\sim$ (cf. [45, Lm. 22]). For the full definition, we refer to the technical report [56]. 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 $\varepsilon$ as next action, we keep the circular shape and call them *neutral states*.

Because of the $\varepsilon$-transitions, 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 collection of local types constitutes a CSM through their semantics.

▶**Definition 2.12 (Implementability [45]).** A global type $G$ is said to be implementable if there exists a deadlock-free CSM $\langle A_p \rangle_{p \in \mathcal{P}}$ such that their languages are the same (protocol fidelity), i.e., $L(G) = L(\langle A_p \rangle_{p \in \mathcal{P}})$. We say that $\langle A_p \rangle_{p \in \mathcal{P}}$ implements $G$.

**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 the technical report [56], we give general descriptions but they are not essential for 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 $\cap$, the projection of a global type $G$ onto a role $r \in \mathcal{P}$ is a local type that is defined as follows:

$$
\sum_{i \in I} p \rightarrow q_i : m_i \cdot G_i |^r := \begin{cases} 
0 |^r := 0 & t |^r := t \\
\varepsilon & \text{if } r = p \text{ and } i \in I
\end{cases}
$$

$$
\left( \sum_{i \in I} p \rightarrow q_i : m_i \cdot G_i |^r \right) |^r := \begin{cases} 
\varepsilon |^r := 0 & \text{if } r \neq t \text{ and } i \in I \\
\varepsilon & \text{otherwise}
\end{cases}
$$

$\mu t.G |^r := \begin{cases} 
\mu t.(G |^r) & \text{if } G |^r \neq t \\
0 & \text{otherwise}
\end{cases}$

$\sum_{i \in I} p \rightarrow q_i : m_i \cdot G_i |^r$

$\varepsilon$

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, i.e. $\mu t.t$, and omit it in this case. We also require to omit $\mu t$ if $t$ is never used in the result.
Intuitively, a projection operator takes the state machine $\text{GAut}(G)$ for a global type $G$ and projects each transition label to the respective alphabet of the role, e.g., $p \to q;m$ becomes $q \triangleleft p;m$ for role $q$. 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 $x$, and $\sqcap$ be a merge operator. We define different cases for the result of $L_1 \sqcap L_2$:

1. $L_1$ if $L_1 = L_2$
2. $\sqcap_{i \in I \cap L_1 \sqcap q;m,L_1,i} \sqcap \sqcap_{i \in I \cap L_2 \sqcap q;m,L_2,i}$
3. $\mu_{L_1,L_2} \sqcap_{i \in I \cap L_2 \sqcap q;m,L_2,i}$

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 $\sqcap$ [23] can apply all Cases (1). The semi-full merge $\sqcap$ [64] can apply Cases (1) and (2). The full merge $\sqcap$ [54] can apply all Cases (1), (2), and (3).

We will also consider the availability merge operator $\sqcap$ by Majumdar et al. [45] which builds on the full merge operator but generalises Case (2) to allow sender-driven 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 $G$ be some global type and let plain merge $\sqcap$, semi full merge $\sqcap$, full merge $\sqcap$, or availability merge $\sqcap$ be the merge operator $\sqcap$. If $G$ is defined for each role $p$, then the CSM $\{L\text{Aut}(G)^p\}_{p \in \mathcal{P}}$ implements $G$.

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. [23] show that subject reduction entails protocol fidelity and progress while increase, in turn, entails deadlock freedom. Subject reduction has been proven for the plain merge operator [23, Thm. 1] and the semi-full operator [64, Thm. 1]. Scalas and Yoshida pointed out that several versions of classical projection with the full merge are flawed [54, 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, correctness was proven by Majumdar et al. [45, Thm. 16].

**Example 3.4 (Projection without merge / Collapsing erasure).** In the introduction, we considered $G_{2BP}$ and the FSM for its semantics in Figure 1a. We projected (without merge) onto seller $s$ to obtain the FSM in Figure 1b. In general, we also collapse neutral states with a single $\varepsilon$-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: $[p \to q;?] \rightsquigarrow \varepsilon$.

**On the Structure of $\text{GAut}(G)$**

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 $\varepsilon$, 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 $\text{lvl}: Q \to \mathbb{N}$ such that, for every transition $q \xrightarrow{z} q' \in \delta$, one of the two holds:
(a) \( \text{lvl}(q) > \text{lvl}(q') \), or
(b) \( x = \varepsilon \) and there is a run from the initial state \( q_0 \) (without going through \( q \)) to \( q' \) which can be completed to reach \( q_0 \to \ldots \to q_n \) as a run with \( q_n = q' \) and \( q \neq q_i \) for every \( 0 \leq i \leq n \), and the run can be extended to \( q_0 \to \ldots \to q_n \to \ldots \to q_{n+m} \) with \( q_{n+m} = q \).

Then, the state \( q' \) is called ancestor of \( q \).

We call the first (a) kind of transition forward transition while the second (b) 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 \( q' \), \( \{q \mid q \xrightarrow{} q' \in \delta \land \text{lvl}(q) > \text{lvl}(q')\} \leq 1 \). The state machine \( A \) is dense if, for every \( q \xrightarrow{} q' \in \delta \), the transition label \( x \) is \( \epsilon \) implies that \( q \) has only one outgoing transition. Last, the cone of \( q \) is all states \( q' \) which are reachable from \( q \) and have a smaller level than \( q \), i.e., \( \text{lvl}(q) > \text{lvl}(q') \).

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

For both, the only forward \( \varepsilon \)-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 supports. 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 it constitutes a negative example for this merge operator. We focus on role \( r \) when projecting. Thus, rejected mostly means that there is (at least) no projection onto \( r \). 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:

\[
\mu t. + \begin{cases} 
  p \to q: l. (q \to r: l. 0 + q \to r: r. t) \\
  p \to q: r. (q \to r: l. 0 + q \to r: r. t)
\end{cases}.
\]

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 \( r \) is not involved in the initial choice. The plain merge operator can resolve this non-determinism since both cones of \( q_1 \) 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 \( r \) which is the result of the (syntactic) plain merge: \( \mu t. (q?l.0 \& q?r.t) \).

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 \( q \to r: l \) and \( q \to r: r \) 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.
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 3.8 (Negative example for plain merge). We consider the following simple implementable global type where the choice by $p$ is propagated by $q$ to $r$: $\mu t \cdot \{ p \to q.l, \ q \to r.l, 0 + q \to r.m, 0 \}$. 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 this implementable global type: $\mu t \cdot \{ p \to q.l, \ q \to r.l, 0 + q \to r.m, 0 \}$. 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.

Example 3.8 (Negative example for plain merge). We consider the following simple implementable global type where the choice by $p$ is propagated by $q$ to $r$: $\mu t \cdot \{ p \to q.l, \ q \to r.l, 0 + q \to r.m, 0 \}$. 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 this implementable global type: $\mu t \cdot \{ p \to q.l, \ q \to r.l, 0 + q \to r.m, 0 \}$. 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 $r$ 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 $r \leftarrow q?l$ and $r \leftarrow q?r$ 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 $r \leftarrow q?m$ which is common to both states. We recursively apply the semi-full merge and, with Case (1), observe that the result $q_{15}$ 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: $\mu t \cdot (q?l.0 & q?m.0 & q?r.t)$.
Asynchronous MST Implementability is Decidable – Lessons Learned from MSCs

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.

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:

\[
\begin{align*}
& p \rightarrow q \cdot \mu t_1. \ (q \rightarrow r \cdot l \cdot q \rightarrow p \cdot l \cdot t_1 + q \rightarrow r \cdot m \cdot q \rightarrow p \cdot m 
& \quad + q \rightarrow r \cdot r \cdot q \rightarrow p \cdot r \cdot t_2)
\end{align*}
\]

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 \(r < q \langle m \) 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 \( q_{14}' \), it applies Case (1) while \( q_{14}' \) 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: \( \mu t_1. \ (q \langle l \cdot t_1 \& q \langle m \cdot 0 \& q \langle r \cdot t_2) \).

Example 3.11 (Negative example for full merge). We consider a simple implementable global type where \( p \) propagates its decision to \( r \) in the top branch while \( q \) propagates it in the bottom branch:

\[
\begin{align*}
& p \rightarrow q \cdot l \cdot p \rightarrow r \cdot l \cdot 0
& \quad + p \rightarrow q \cdot r \rightarrow r \cdot 0'.
\end{align*}
\]

It is illustrated in Figure 3c. This cannot be projected onto \( r \) by the full merge operator for which all receive events need to have the same sender.

Remark 3.12 (On sender-driven choice). Majumdar et al. [45] proposed a classical projection operator that overcomes 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. Majumdar et al. 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 all classical projection operators. For details, we refer to their work [45].

Shortcomings of Classical Projection/Merge Operators

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 \( a \rightarrow s \cdot n o \) with which buyer \( a \) notifies seller \( s \) that they will not buy the item:

\[
\begin{align*}
& \mu t. + \{ a \rightarrow s \cdot q u e r y, s \rightarrow a \cdot p r i c e, \ (a \rightarrow b \cdot s p l i t. (b \rightarrow a \cdot y e s) \rightarrow a \cdot b \cdot b u y. t \rightarrow a \cdot n o. t \) + a \rightarrow b \cdot c a n c e l. t \}
\rightarrow a \rightarrow s \cdot d o n e. a \rightarrow b \cdot d o n e. 0
\end{align*}
\]
This global type cannot be projected onto seller $a$. 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 for seller $a$:

$$
\mu t. \left\{ \begin{array}{ll}
\text{a?query. } \mu t_1. \text{a!price. (a?buy. } t_1 \& \text{a?query. } t_2 \& \text{a?done. } 0) \\
\text{a?done.} 
\end{array} \right.
$$

The local type has two recursion variable binders while the global type only has one. 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: $G_{\text{2BPWS}} := \{ a \to s. \text{login. } G_{\text{2BP}} \} \cup \{ a \to s. \text{subscribe. } a \to b. \text{subscribed. } G_{\text{2BP}} \}$. The merge operator needs to merge a recursion variable binder $\mu t$ with the external choice $b \otimes a?\text{subscribed}$. Still, there is a local type $L_b$ for $b$ such that $L(L_b) = L(G_{\text{2BPWS}})\upharpoonright_{L_b}$:

$$
L_b := \left\{ \begin{array}{ll}
\text{a?split. (a?yes. L(t) \& a?no. L(t))} \\
\text{a?cancel. L(t)} \\
\text{a?done. 0} \\
\text{a?subscribed. L(t)}
\end{array} \right.
$$

where $L(t) := \mu t. \left\{ \begin{array}{ll}
\text{a?split. (a?yes. t \& a?no. t)} \\
\text{a?cancel. t} \\
\text{a?done. 0}
\end{array} \right.$

In fact, one can also rely on the fact that buyer $a$ will comply with the intended protocol. Then, it suffices to introduce one recursion variable $t$ in the beginning and substitute every $L(\cdot)$ with $t$, yielding a local type $L'_b$ with $L(L'_b) \subseteq L(L_b)$.

Similarly, classical projection operator cannot handle global types where choices can be disambiguated with semantic properties, e.g., counting modulo a constant. Scalas and Yoshida [54] also 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 [54, 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):

$$
G_{\text{2BPWR}} := a \to s. \text{query. } s \to a. \text{price. } \mu t. \left\{ \begin{array}{ll}
\text{a \to b. split. (b \to a. \text{yes. } a \to s. \text{buy. } 0 + b \to a. \text{no. } t)} \\
\text{a \to b. cancel. a \to s. } \text{no. } 0
\end{array} \right.
$$

These shortcomings have been addressed by some non-classical approaches. For example, Scalas and Yoshida [54] employ model checking while Castagna et al. [16] 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 sender-driven choice, is decidable. We answer this question positively for the more general case of sender-driven choice.

## 4 Implementability for Global Types from MSTs is Decidable

In this section, we show decidability of the implementability problem for global types with sender-driven choice, using results from the domain of message sequence charts. We introduce high-level message sequence charts (HMSCs) and recall an HMSC encoding for global types. In general, implementability for HMSCs is undecidable but we show that global types, when encoded as HMSCs, belong to a class of HMSCs for which implementability is decidable.
Asynchronous MST Implementability is Decidable – Lessons Learned from MSCs

4.1 High-level Message Sequence Charts

Our definitions of (high-level) message sequence charts follow work by Genest et al. [30] and Stutz and Zufferey [57]. If reasonable, we adapt terminology to the MST setting.

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

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

We visually highlight the elements of an MSC in Figure 5.

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

- $e \preceq_M e'$ PROC $s \in S$ SND-RCV $e \preceq_M e'$ REFL $e \preceq_M e'$ TRANS

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

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

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). A high-level message sequence chart (HMSC) is a 5-tuple $(V, E, v^T, V^T, \mu)$ where $V$ is a finite set of vertices, $E \subseteq V \times V$ is a set of directed edges, $v^T \in V$ is an initial vertex, $V^T \subseteq V$ is a set of terminal vertices, and $\mu : V \rightarrow M$ 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 the technical report [56]. 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 [3]).** An HMSC $H$ is said to be safely realisable if there exists a deadlock-free CSM $\{ A_p \}_{p \in P}$ such that $L(H) = L(\{ A_p \}_{p \in P})$.

**Encoding Global Types from MSTs as HMSCs**

Stutz and Zufferey [57, Sec. 5.2] provide a formal encoding $H(-)$ from global types to HMSCs. We refer to the technical report [56] for the definition. We adapt their correctness result to our setting. In particular, our semantics of $G$ use the closure operator $C^\ast(-)$ while they 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 [57, Lm. 5].

**Theorem 4.4.** Let $G$ be a global type. Then, the following holds: $L(G) = L(H(G))$.

### 4.2 Implementability is Decidable

We introduce a mild assumption for global types. Intuitively, we require that every run of the protocol can always terminate but does not need to. Basically, this solely rules out global types that have loops without exit (cf. Example 4.20). In practice, it is reasonable to assume a mechanism to terminate a protocol for maintenance for instance. Note that this assumption constitutes a structural property of a protocol and no fairness condition on runs of the protocol.

**Assumption (0-Reachable).** We say a global type $G$ is 0-reachable if every prefix of a word in its language can be completed to a finite word in its language. Equivalently, we require that the vertex for the syntactic subterm 0 is reachable from any vertex in $H(G)$.

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 [2, Thm. 13]. Therefore, approaches center around checking safe realisability of HMSC languages and establishing conditions on HMSCs that entail safe realisability.

**Definition 4.5 (Canonical candidate implementation [2]).** Given an HMSC $H$ and a role $p$, let $A_p' = (Q', \Sigma_p, \delta', q'_0, F')$ be a state machine with $Q' := \{ q_w \mid w \in \text{pref}(L(H)\downharpoonright_{\Sigma_p}) \}$, $F' := \{ q_w \mid w \in L_{\text{fin}}(H)\downharpoonright_{\Sigma_p} \}$, and $\delta'(q_w, x, q_{wx})$ for $x \in \Sigma_{\text{async}}$. The resulting state machine $A_p'$ is not necessarily finite so $A_p'$ is determinised and minimised which yields the FSM $A_p$. We call $\{ A_p \}_{p \in P}$ the canonical candidate implementation of $H$.

Intuitively, the intermediate state machine $A_p'$ constitutes a tree whose maximal finite paths give $L(H)\downharpoonright_{\Sigma_p} \cap \Sigma_p^*$. 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. [3, Proof of Thm. 3].

**Definition 4.6 (Projection by Erasure).** Let $p$ be a role and $M = (N, p, f, l, (\leq_p)_{p \in P})$ be an MSC. We denote the set of nodes of $p$ with $N_p := \{ n \mid p(n) = p \}$ and define a two-ary next-relation on $N_p$: $\text{next}(n_1, n_2)$ iff $n_1 \leq n_2$ and there is no $n'$ with $n_1 \leq n' \leq n_2$. We define the projection by erasure of $M$ on to $p$: $M\downharpoonright_p = (Q_M, \Sigma_p, \delta_M, q_{M,0}, (q_{M,f}))$ with
Asynchronous MST Implementability is Decidable – Lessons Learned from MSCs

\[ Q_M := \{ q_n \mid n \in N \} \sqcup \{ q_{M,0} \} \sqcup \{ q_{M,f} \} \quad \text{and} \]
\[ \delta_M := \{ q_{M,0} \xrightarrow{e} q_{n_1} \mid \forall n_2, n_1 \leq n_2 \} \sqcup \{ q_{n_1} \xrightarrow{(n_1)} q_{n_2} \mid \text{next}(n_1, n_2) \} \sqcup \{ q_{n_2} \xrightarrow{(n_2)} q_{M,f} \mid \forall n_1, n_1 \leq n_2 \} \]

where \( \sqcup \) denotes disjoint union. Let \( H = (V, E, v^1, V^T, \mu) \) be an HMSC. We construct the projection by erasure for every vertex and identify them with the vertex, i.e., \( Q_v \) instead of \( Q_{\mu(v)} \). We construct an auxiliary FSM \( (Q'_H, \Sigma_p, \delta'_H, q'_{H,0}, F'_H) \) with \( Q'_H = \bigcup_{v \in V} Q_v \), \( \delta'_H = \bigcup_{v \in V} \delta_v \sqcup \{ q_{v,0} \xrightarrow{f} q_{v,2,0} \mid (v_1, v_2) \in E \} \), \( q'_{H,0} = q_{v,0} \), and \( F'_{H} = \bigcup_{v \in V} q_{v,f} \). We determinise and minimise \( (Q'_H, \Sigma_p, \delta'_H, q'_{H,0}, F'_H) \) to obtain \( H_\downarrow_p := (Q_H, \Sigma_p, \delta_H, q_{H,0}, F_H) \), which we define to be the projection by erasure of \( H \) onto \( p \). The CSM formed from the projections by erasure \( \{ H_\downarrow_p \}_{p \in P} \) is called erasure candidate implementation.

\[ \Box \text{Lemma 4.7 (Correctness of Projection by Erasure).} \quad \text{Let} \quad H \text{ be an HMSC,} \quad p \text{ be a role, and} \quad H_\downarrow_p \text{ be its projection by erasure. Then, the following language equality holds:} \quad \mathcal{L}(H_\downarrow_p) = \mathcal{L}(H)_\downarrow \Sigma_p. \]

The proof is straightforward and can be found in the technical report [56]. From this result and the construction of the canonical candidate implementation, it follows that the projection by erasure admits the same finite language.

\[ \Box \text{Corollary 4.8.} \quad \text{Let} \quad H \text{ be an HMSC,} \quad p \text{ be a role,} \quad H_\downarrow_p \text{ be its projection by erasure, and} \quad A_p \text{ be the canonical candidate implementation. Then, it holds that} \quad \mathcal{L}_{\text{fin}}(H_\downarrow_p) = \mathcal{L}_{\text{fin}}(A_p). \]

The projection by erasure can be computed effectively and is 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.

\[ \Box \text{Theorem 4.9.} \quad \text{Let} \quad G \text{ be a global type and} \quad \{ H(G)_\downarrow_p \}_{p \in P} \text{ be its erasure candidate implementation. If} \quad \mathcal{L}_{\text{fin}}(G) \text{ is implementable}^3, \text{ then} \quad \{ H(G)_\downarrow_p \}_{p \in P} \text{ is deadlock-free and} \quad \mathcal{L}_{\text{fin}}(\{ H(G)_\downarrow_p \}_{p \in P}) = \mathcal{L}_{\text{fin}}(G). \]

This result does only account for finite languages so we extend it for infinite sequences. For both, the proof can be found in the technical report [56].

\[ \Box \text{Lemma 4.10 ("Finite implementation" generalises to infinite language for 0-reachable global types).} \quad \text{Let} \quad G \text{ be a 0-reachable global type and} \quad \{ A_p \}_{p \in P} \text{ be an implementation for} \quad \mathcal{L}_{\text{inf}}(G). \quad \text{Then, it holds that} \quad \mathcal{L}_{\text{inf}}(\{ A_p \}_{p \in P}) = \mathcal{L}_{\text{inf}}(G), \quad \text{and, thus,} \quad \mathcal{L}(\{ A_p \}_{p \in P}) = \mathcal{L}(G). \]

\[ \Box \text{Corollary 4.11.} \quad \text{Let} \quad G \text{ be a 0-reachable implementable global type. Then, the erasure candidate implementation} \quad \{ H(G)_\downarrow_p \}_{p \in P} \text{ implements} \quad G. \]

So far, we have shown that, if \( G \) is implementable, the erasure candidate implementation for its HMSC encoding \( H(G) \) implements \( G \). For HMSCs, this is undecidable in general [44]. We show that, because of their syntactic restrictions on choice, global types fall into the class of globally-cooperative HMSCs for which implementability is decidable.

\[ \Box \text{Definition 4.12 (Communication graph [31]).} \quad \text{Let} \quad M = (N, p, f, l, (\leq_p)_{p \in P}) \text{ be an MSC. The communication graph of} \quad M \text{ is a directed graph with node} \quad p \text{ for every role} \quad p \text{ that sends or receives a message in} \quad M \text{ and edges} \quad p \rightarrow q \text{ if} \quad M \text{ contains a message from} \quad p \text{ to} \quad q, \text{ i.e., there is} \quad e \in N \text{ such that} \quad p(e) = p \text{ and} \quad p(f(e)) = q. \]

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

\[ ^3 \text{Implementability is lifted to languages as expected.} \]
**Definition 4.13** (Globally-cooperative HMSCs [31]). An HMSC $H = (V, E, v^l, V^T, \mu)$ 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, i.e., all nodes are connected if every edge is considered undirected.

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

**Theorem 4.14** (Thm. 3.7 [44]). Let $H$ be a globally-cooperative HMSC. Restricted to its finite language $L_{\text{fin}}(H)$, safe realisability is EXPSPACE-complete.

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

The proof can be found in the technical report [56] 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 but not globally cooperative). HMSC $H_{\text{ing}}$ in Figure 6a is implementable but neither globally-cooperative nor representable with a global type. In the first loop, $p$ sends a message $m$ to $q$ while $r$ sends a message $m$ to $s$ so the communication graph is not weakly connected. In the second loop, 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 variant of the protocol without the second loop, any candidate 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 second loop can make up for such executions so any execution has a path in $H_{\text{ing}}$. The CSM in Figure 6b implements $H_{\text{ing}}$. In the technical report [56], we explain in detail why there is a path in $H_{\text{ing}}$ for any trace of the CSM and how to modify the example not to have final states with outgoing transitions.

**Theorem 4.17.** Checking implementability of 0-reachable global types with sender-driven choice is in EXPSPACE.

**Proof.** Let $G$ be a 0-reachable global type with sender-driven choice. We construct $H(G)$ from $G$ and check if it is globally-cooperative. For this, we apply the coNP-algorithm by Genest et al. [31] which is based on guessing a subgraph and checking its communication graph. If $H(G)$ is not globally cooperative, we know from Lemma 4.15 that $G$ is not implementable. If $H(G)$ is globally cooperative, we check safe realisability for $H(G)$. By Theorem 4.14, this is in EXPSPACE. If $H(G)$ is not safely realisable, it trivially follows that $G$ is not implementable. If $H(G)$ is safely realisable, $G$ is implemented by the erasure candidate implementation with Theorem 4.9 and Lemma 4.10.
Thus, the implementability problem for global types with sender-driven choice is decidable.

**Corollary 4.18.** Let $G$ be a 0-reaching global type with sender-driven choice. It is decidable whether $G$ is implementable and there is an algorithm to obtain its implementation.

**Remark 4.19 (Progress).** The property deadlock freedom is sometimes also studied as progress – in the sense that a system should never get stuck. For infinite executions, however, a role could starve in a non-final state by waiting for a message that is never sent [16, Sec. 3.2]. Thus, Castagna et al. [16] consider a stronger notion of progress (Def. 3.3: live session) which requires that every role could eventually reach a final state. Our results also apply to this stronger notion of progress, which entails that any sent message can eventually be received. The notion only requires it to be possible but we can ensure that no role starves in a non-final state in two ways. First, we can impose a (strong) fairness assumption – as Castagna et al. [16]. 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 theory, one can think of protocols for which the 0-Reachability-Assumption (p.15) 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 4.20.** Consider the following global type: $G = \mu t.p \rightarrow q:m.x \rightarrow s:m.t$. This is basically the protocol that consists only of the first loop of $H_{ing}$ (Example 4.16). It describes an infinite execution with two pairs of roles that independently send and receive messages. This can be implemented in an infinite setting but the loop can never be exited due to the lack of synchronisation, breaking protocol fidelity upon termination.

**Expressiveness of Local Types.** Local types also have a distinct expression for termination: 0. Thus, if one considers the FSM of a local type, every final state has no outgoing transition. Our proposed algorithm might yield FSMs for which this is not the case. However, the language of such an FSM cannot be represented as local type since both our construction and FSMs for local types are deterministic. The latter are also ancestor-recursive, free of intermediate recursion, non-merging and dense (Proposition 3.6). For FSMs from our procedure, this is not the case but the ones without final states with outgoing transitions could possibly be transformed to local types, making subtyping techniques applicable. One could also study subtyping for FSMs as local specifications. We leave both for future work.

**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 [44]. This hardness result does not carry over for the HMSC encoding $H(G)$ of a global type $G$. 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, yielding an EXPSPACE-algorithm. In this section, we consider further restrictions on HMSCs to obtain algorithms with better complexity for global types. First, we transfer the algorithms for $I$-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. Third, we present a variant of the implementability problem. It can make unimplementable global types implementable without changing a protocol’s structure. From now on, we may use the term implementability for HMSCs instead of safe realisability.

\section*{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 I-closed HMSCs \cite[Thm. 3.7]{44}. These require the language of an HMSC to be closed with regard to an independence relation $I$, where, intuitively, two interactions are independent if there is no role which is involved in both. Implementability for I-closed HMSCs is PSPACE-complete \cite[Thm. 3.6]{44}. 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 \cite{44} 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 $I$ on the alphabet $\Sigma_{\text{sync}}$.

\begin{definition}[Independence relation $I$]
We define the independence relation $I$ on $\Sigma_{\text{sync}}$:

\begin{equation}
I := \{(p \rightarrow q : m, r \rightarrow s : m') \mid \{p, q\} \cap \{r, s\} = \emptyset\}.
\end{equation}

We lift this to an equivalence relation $\equiv_I$ on words as its transitive and reflexive closure:

\begin{equation}
\equiv_I := \{(u, x_1, x_2, w, u, x_2, x_1, w) \mid u, w \in \Sigma_{\text{sync}}^* \text{ and } (x_1, x_2) \in I\}.
\end{equation}

We define its closure for language $L \subseteq \Sigma_{\text{sync}}^*$:

\begin{equation}
C^{\equiv_I}(L) := \{u \in \Sigma_{\text{sync}}^* \mid \exists w \in L \text{ with } u \equiv_I w\}.
\end{equation}

\end{definition}

\begin{definition}[I-closed global types]
Let $G$ be a global type $G$. We say $G$ is I-closed if $L_{\text{fin}}(\text{GAut}(G)) = C^{\equiv_I}(L_{\text{fin}}(\text{GAut}(G)))$.
\end{definition}

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

\begin{example}
The global type $G_{\text{2BP}}$ is I-closed. Buyer a is involved in every interaction. Thus, for every two consecutive interactions, there is a role that is involved in both.
\end{example}

\begin{algorithm}[Checking if $G$ is I-closed]
Let $G$ be a global type. We construct the state machine $\text{GAut}(G)$. We need to check every consecutive occurrence of elements from $\Sigma_{\text{sync}}$ for words from $L(\text{GAut}(G))$. For binder states, incoming and outgoing transition labels are always $\varepsilon$. 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 $x, y \in \Sigma_{\text{sync}}$ of every combination of incoming and outgoing transition of $q$. We check if $x \equiv_I y$. If this is true for all $x$ and $y$, we return true. If not, we return false.
\end{algorithm}

\begin{lemma}
A global type $G$ is I-closed iff Algorithm 1 returns true.
\end{lemma}
Asynchronous MST Implementability is Decidable – Lessons Learned from MSCs

The proof can be found in the technical report [56]. This shows that the presented algorithm can be used to check $I$-closedness. The algorithm considers every state and all combinations of transitions leading to and from it.

**Proposition 5.5.** For global type $G$, checking if $G$ is $I$-closed is in $O(|G|^2)$.

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

**Example 5.6.** Let us consider the following global type for some $n \in \mathbb{N}$.

\[
\mu t. + \begin{cases} 
p \rightarrow q_0 : m_0, q_0 \rightarrow r_0 : m_0, r_0 \rightarrow q_0 : m_0, 0 \\
p \rightarrow q_1 : m_1, q_1 \rightarrow r_1 : m_1, r_1 \rightarrow q_1 : m_1, t \\
\vdots \\
p \rightarrow q_n : m_n, q_n \rightarrow r_n : m_n, r_n \rightarrow s_n : m_n, t
\end{cases}
\]

It is obvious that $(p \rightarrow q_i : m_i, q_i \rightarrow r_i : m_i) \notin I$ and $(q_i \rightarrow r_i : m_i, r_i \rightarrow s_i : m_i) \notin I$ for every $i$. Because of the recursion, we need to check if $(r_i \rightarrow s_i : m_i, p \rightarrow q_j : m_j)$ is in $I$ for every $0 \neq i \neq j$. This might lead to a quadratic number of checks.

If a global type $G$ is $I$-closed, we can apply the results for its $I$-closed HMSC encoding $H(G)$, for which checking implementability is in PSPACE. With Corollary 4.11, the projection by erasure implements $G$.

**Corollary 5.7.** Checking implementability of 0-reachable, $I$-closed global types with sender-driven choice is in PSPACE.

**Example 5.8.** This implementable global type is not $I$-closed: $p \rightarrow q : m, r \rightarrow s : m, 0$.

**Detecting Non-local Choice in HMSCs**

For HMSCs, there are no restrictions on branching. Similar to choice for global types, the idea of imposing restrictions on choice was studied for HMSCs [8, 50, 48, 34, 31]. We refer to [45] 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. [26] centers around the idea of non-local choice. Intuitively, non-local choice yields scenarios that make 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 [6] and reformulated the closure conditions for safe realisability by Alur et al. [2]. In particular, they provide a definition that 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 $O(k^2 \cdot |P| + n \cdot |P|)$, 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.

**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 while preserving the structure of the protocol. It can also be used if a global type is rejected by a projection operator or the run time of the previous algorithms is not acceptable.

Definition 5.9 (Payload implementability). Let $L$ be a language with message alphabet $V_1$. We say that $L$ is payload implementable if there is a message alphabet $V_2$ for a deadlock-free CSM $\{A_p\}_{p \in P}$ with $A_p$ over $\{p \triangleright q m, p \triangleleft q?m \mid q \in P, m \in V_1 \times V_2\}$ such that its language is the same when projecting onto the message alphabet $V_1$, i.e., $\mathcal{L}_1(L) = \mathcal{L}(\{A_p\}_{p \in P})\downarrow V_1$, where $(p \triangleright q(m_1, m_2))\downarrow V_1 := p \triangleright q m_1$ and $(q \triangleleft p?(m_1, m_2))\downarrow V_1 := q \triangleleft p?m_1$ and is lifted to words and languages as expected.

The finite language $L_{\text{fin}}(\mathcal{H})$ of a local HMSC $\mathcal{H}$ is always payload implementable with a deadlock-free CSM of linear size.

Definition 5.10 (Local HMSCs [31]). Let $H = (V, E, V^l, V^r, \mu)$ be an HMSC. We call $H$ local if $\mu(v')$ has a unique minimal event and there is a function root: $V \rightarrow P$ such that for every $(v, u) \in E$, it holds that $\mu(u)$ has a unique minimal event $e$ and $e$ belongs to root$(v)$, i.e., for $\mu(u) = (N, p, f, l, (\leq_p)_{p \in P})$, we have that $p(e) = \text{root}(v)$ and $e \leq e'$ for every $e' \in N$.

Proposition 5.11 (Prop. 21 [31]). For any local HMSC $\mathcal{H}$, $L_{\text{fin}}(\mathcal{H})$ is payload implementable.

With Lemma 4.10, we can use the implementation of a local $H(G)$ for a 0-reachable global type $G$.

Corollary 5.12. Let $G$ be a 0-reachable for which $H(G)$ is local. Then, $G$ can be implemented with a CSM of linear size.

The algorithm to construct a deadlock-free CSM [31, 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: $\mu_1 + \{p \triangleright q m_1, r \rightarrow s : m_2. t\}$. However, is easy to see that it is not payload implementable since there is no interaction between $p$ and $r$, which decides whether to stay in the loop or not, and $s$. Thus, we cannot simply check whether $H(G)$ 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 $H(G)$ is local – directly on $G\text{Aut}(G)$). Let $G$ be a global type $G$. We consider the finite trace $w'$ of every longest branch-free, loop-free and non-initial run in the state machine $G\text{Aut}(G)$. We split the (synchronous) interactions into asynchronous events: $w = \text{split}(w') = w_1 \ldots w_n$. We need to check if there is $u \sim w$ with $u = u_1 \ldots u_n$ such that $w_1 \neq w_1$. For this, we can construct an MSC for $w$ [30, Sec. 3.1] and check if there is a single minimal event. For this, we use the implementation of $G\text{Aut}(G)$ as a MSC. Because MSCs are closed under $\sim$ [57, Lm. 5]. If the MSC of every trace $w'$ has a single minimal event, we return true. If not, we return false.

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

Proposition 5.13. For a global type $G$, Algorithm 2 returns true if $H(G)$ is local.

Ben-Abdallah and Leue [8] 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 that accepts the same language with a quadratic blow-up [31]. 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. 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 send and receive 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 that our model does not account for. Therefore, 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 \( p \) distributes independent tasks to different roles in rounds:

\[
G_{TC} = \mu t. \begin{cases} p \to q_1 \text{ task.} \ldots \to q_n \text{ task.} \quad q_1 \to p \text{ result.} \ldots \quad q_n \to p \text{ result.} \quad t \\ p \to q_1 \text{ done.} \ldots \to q_n \text{ done.} 0 \end{cases}
\]

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

We generalise the indistinguishability relation \( \approx \) accordingly.

Definition 6.2 (Intra-role indistinguishability relation \( \approx \)). We define a family of intra-role indistinguishability relations \( \approx_i \subseteq \Sigma^* \times \Sigma^* \), for \( i \geq 0 \) as follows. For all \( w, u \in \Sigma^* \), \( w \approx_i u \) entails \( w \approx_1 u \). For \( i = 1 \), we define: if \( q \neq r \), then \( w.p \approx q \text{?} m.p \approx q \text{?} m'.u \approx_1 w.p \approx q \text{?} m'.p \approx q \text{?} m.u \). Based on this, we define \( \approx \) analogously to \( \approx_1 \). Let \( w, w' \), \( w'' \) be words s.t. \( w \approx_i w' \) 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 \( u \approx v \) if there is \( w \in \Sigma^* \) such that \( u.w \approx v \). Observe that \( u \approx v \) iff \( u \approx_0 v \) and \( v \approx_0 u \). We extend \( \approx \) to infinite words and languages as \( \approx_\infty \).

Definition 6.3 (Implementability w.r.t. \( \approx \)). A global type \( G \) is implementable with regard to \( \approx \) if there exists a deadlock-free CSM \( \{ A_p \}_{p \in P} \) such that (i) \( \mathcal{L}(G) \subseteq C^\approx(\mathcal{L}(\{ A_p \}_{p \in P})) \) and (ii) \( C^\approx(\mathcal{L}(G)) = C^\approx(\mathcal{L}(\{ A_p \}_{p \in P})) \). We say that \( \{ A_p \}_{p \in P} \approx \)-implements \( G \).

In this section, we emphasise the indistinguishability relation, e.g., \( \approx \)-implementable. We could have also followed the definition of \( \approx \)-implementability and required \( C^\approx(\mathcal{L}(G)) = \mathcal{L}(\{ A_p \}_{p \in P}) \). This, however, requires the CSM to be closed under \( \approx \). In general, this might not be possible with finitely many states. In particular, if there is a loop without any send events for a role, the labels in the loop would introduce an infinite closure if we require that \( C^\approx(\mathcal{L}(G)) \mathcal{L}(A_p) \).

Example 6.4. We consider a variant of \( G_{TC} \) from Example 6.1 with \( n = 2 \) where \( q_1 \) and \( q_2 \) send a log message to \( r \) after receiving the task and before sending the result back:

\[
G_{TClog} := \mu t. \begin{cases} p \to q_1 \text{ task.} \quad p \to q_2 \text{ task.} \quad q_1 \to r \text{ log.} \quad q_2 \to r \text{ log.} \quad q_1 \to p \text{ result.} \quad q_2 \to p \text{ result.} \quad t \\ p \to q_1 \text{ done.} \quad p \to q_2 \text{ done.} 0 \end{cases}
\]

There is no FSM for \( r \) that precisely accepts \( C^\approx(\mathcal{L}(G_{TClog})) \mathcal{L}(\Sigma_r) \). If we rely on the fact that \( q_1 \) and \( q_2 \) send the same number of log-messages to \( r \), we can use an FSM \( A_r \) with a single state (both initial and final) with two transitions: one for the log-message from \( q_1 \) and \( q_2 \)}.
then, we require that the solution starts with the first pair so there exists no trivial solution and $x$ global type where choosing a single pair is more concise than all possible ones. Our encoding is the following to directly reduce from PCP but the reduction from MPCP is more concise. Intuitively, match starts with a specific pair – in our case we choose the pair with index $c$-

Proposition 6.5. Let $G$ be a global type that is $\sim$-implemented by the CSM $\{A_p\}_{p \in P}$. Then, $\{A_p\}_{p \in P}$ also $\sim$-implements $G$.

For instance, the erasure candidate implementation is a $\sim$-implementation as well as a $\approx$-implementation for the task coordination protocol $G_{TC}$ from Example 6.1. 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, related by $\approx$ though, in different branches it cannot distinguish (yet).

Example 6.6 ($\approx$-implementable but not $\sim$-implementable). Let us consider the following global type: $(p \rightarrow q: l. p \rightarrow r: m. q \rightarrow r: m. 0) + (p \rightarrow q: r. q \rightarrow r: m. p \rightarrow r: m. 0)$. This cannot be $\sim$-implemented because $r$ would need to know about the choice to receive the messages from $p$ and $q$ in the correct order. However, it is $\approx$-implementable. The FSMs for $p$ and $q$ can be obtained with projection by erasure. For $r$, we can have an FSM that only accepts $r \triangleright q?m. r \triangleleft q?m$ but also an FSM which accepts $r \triangleleft q?m. r \triangleright q?m$ in addition. Note that $r$ does not learn the choice in the second FSM even if it branches. Hence, it would not be 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) [53] to the problem of checking implementability with regard to $\approx$. An instance of PCP over an alphabet $\Delta$, $|\Delta| > 1$, 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 $\Delta$, also called tile sets. 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 $u_{i_1} \ldots u_{i_k} = v_1 \ldots v_k$. To be precise, we present a reduction from the modified PCP (MPCP) [55, 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 from 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 $c-x$ indicates choosing tile set $x$, and message $ack-x$ indicates acknowledging the tile set $x$:

$$G_{MPCP} := \{ G(u, r \rightarrow p: ack-u, 0) \} \cup \{ G(v, r \rightarrow p: ack-v, 0) \}$$

$$G(x, X) := p \rightarrow q: o-x. p \rightarrow q: 1. p \rightarrow r: 1. q \rightarrow r: [x_i]. \mu t. + \{ p \rightarrow q: 1. p \rightarrow r: 1. q \rightarrow r: [x_i]. t : p \rightarrow q: n. p \rightarrow r: n. q \rightarrow r: [x_n]. t \}$$

$$\begin{align*}
 p \rightarrow q: d. p \rightarrow r: d. q \rightarrow r: d. X
\end{align*}$$

ECCOOP 2023
Asynchronous MST Implementability is Decidable – Lessons Learned from MSCs

The HMSC encoding $H(G_{\text{MPCP}})$ is illustrated in Figure 7. Intuitively, $x$ eventually needs to know which branch was taken in order to match $\text{ack}$-$x$ with $c$-$x$ from the beginning. However, it can only know if there is no solution to the MPCP instance. In the full proof in technical report [56], we show that $G_{\text{MPCP}}$ is $\approx$-implementable iff the MPCP instance has no solution.

▶ Theorem 6.7. Checking implementability with regard to $\approx$ for $0$-reachable 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 $\{\{A_p\}_{p \in P}\}$ such that the following holds: (i) $L(H) \subseteq C^\approx(L(\{\{A_p\}_{p \in P}\}))$ and (ii) $C^\approx(L(H)) = C^\approx(L(\{\{A_p\}_{p \in P}\)))$.

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

It is obvious that a terminal vertex is reachable from every vertex in $H(G_{\text{MPCP}})$. In fact, the HMSC encoding for $G_{\text{MPCP}}$ also satisfies a number of channel restrictions. The HMSC $H(G_{\text{MPCP}})$ is existentially 1-bounded, 1-synchronisable and half-duplex [57]. For details on these channel restrictions, we refer to work by Stutz and Zufferey [57, Sec. 3.1].

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. We leave a detailed analysis for future work.

7 Related Work

In this section, we solely cover related work which was not discussed before.

Multiparty Session Types. Session types originate in process algebra and were first introduced by Honda et al. [35] for binary sessions. For systems with more than two roles, they have been extended to multiparty session types [37]. We explained MST frameworks with classical projection operators. Other approaches do not focus on projection but only apply ideas from MST without the need for global types [54, 43].

Completeness and Sender-driven Choice. Our decidability result applies to global types with sender-driven choice. To the best of our knowledge, the work by Castagna et al. [16] is the only one to attempt completeness for global types with sender-driven choice. However, their definition of completeness is "less demanding then other ones" [16, Abs.]. For one
global type, they accept different implementations that generate different sets of traces [16, Def. 4.1 and Sec. 5.3]. 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. [45] employ a global availability analysis but, as classical projection operator, it suffers from the shortcomings presented in this work. For a detailed overview of MST frameworks with sender-driven choice, we refer to their work [45]. The global types by Castellani et al. [17] specify send and receive events independently and allow to receive from different senders. Dagnino et al. [25] consider similar global types but each term requires to send to a single receiver and to receive from a single sender upon branching through.

On the Synchronous Implementability Problem. We could not find a reference that shows decidability of the implementability problem in a synchronous setting, i.e., without channels. Before giving a proof sketch, let us remark that there are global types that can be implemented synchronously but not asynchronously, e.g., \( p \rightarrow q : l \rightarrow q : l ; 0 + p \rightarrow q : r \rightarrow q : r ; 0 \) because \( q \) can force the right choice by \( r \). We sketch how to prove decidability of the synchronous implementability problem for global types (with sender-driven choice). One defines the synchronous semantics of CSMs and HMSCs as expected. For global types, one uses the independence relation \( I \) (Def. 5.1), which defines reasonable reorderings for synchronous events in a distributed setting, similar to the indistinguishability relation \( \sim \). It is straightforward that the HMSC encoding \( H(\cdot) \) for global types [57] also works for the synchronous setting (cf. Thm. 4.4). Thus, every implementation for \( H(G) \) is also an implementation for \( G \). For the asynchronous setting, we used [2, Thm. 13], which shows that the canonical candidate implementation implements an HMSC if it is implementable. Alur et al. [2, Sec. 8] also considered the synchronous setting. They observe that both Theorem 5 and 8, basis for Theorem 13, stay valid under these modified conditions. Together with our results, the erasure candidate implementation implements a global type if it is implementable. Because of the synchronous semantics, its state space is finite and can be model-checked against the global type, yielding a PSPACE-procedure for the synchronous implementability problem, and thus decidability.

Subtyping and MST Extensions. In this work, we do not distinguish between local types and implementations but use local types directly as implementations. Intuitively, subtyping studies how to give freedom in the implementation while preserving the correctness 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. [24]. 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. [22, 21], Lange and Yoshida [42], and Bravetti et al. [14]. Various extensions to make MST verification applicable to more scenarios were studied: for instance delegation [36, 37, 18], dependent session types [59, 29, 60], parametrised session types [20, 29], gradual session types [39], or dynamic self-adaption [33]. Context-free session types [58, 41] provide a more expressive way to specify protocols. Research on fault-tolerant MSTs [63, 7] investigates ways to weaken the strict assumptions about reliable channels.
Communicating State Machines. The connection of MSTs and CSMs was studied soon after MSTs had been proposed [28]. CSMs are known to be Turing-powerful [13]. Decidable classes have been obtained for different semantics, e.g., half-duplex communication for two roles [19], input-bounded [10], and unreliable/lossy channels [1], as well for restricted communication topology [52, 61]. Similar restrictions for CSMs are existential boundedness [30] and synchronisability [12, 32]. It was shown that global types can only express existentially 1-bounded, 1-synchronisable and half-duplex communication [57] while Bollig et al. [11] established a connection between synchronisability and MSO logic.

High-level Message Sequence Charts. Globally-cooperative HMSCs were independently introduced by Morin [49] as c-HMSCs. Their communication graph is weakly connected. The class of bounded HMSCs [4] 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 [44]. This class was independently introduced as regular HMSCs by Muscholl and Peled [51]. 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 [31, Prop. 4].

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 in general, 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 cannot be implemented. Another key was the extension of the HMSC results to infinite executions. We also 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


