








Complete Multiparty Session Type Projection with Automata

Elaine Li¹ , Felix Stutz²  , Thomas Wies¹ , and Damien Zufferey³ 



¹ New York University, New York, USA
ef19013@nyu.edu, wies@cs.nyu.edu

² Max Planck Institute for Software Systems,
Kaiserslautern, Germany
fstutz@mpi-sws.org

³ SonarSource, Geneva, Switzerland
damien.zufferey@sonarsource.com



Abstract. Multiparty session types (MSTs) are a type-based approach to verifying communication protocols. Central to MSTs is a *projection operator*: a partial function that maps protocols represented as global types to correct-by-construction implementations for each participant, represented as a communicating state machine. Existing projection operators are syntactic in nature, and trade efficiency for completeness. We present the first projection operator that is sound, complete, and efficient. Our projection separates synthesis from checking implementability. For synthesis, we use a simple automata-theoretic construction; for checking implementability, we present succinct conditions that summarize insights into the property of implementability. We use these conditions to show that MST implementability is PSPACE-complete. This improves upon a previous decision procedure that is in EXPSPACE and applies to a smaller class of MSTs. We demonstrate the effectiveness of our approach using a prototype implementation, which handles global types not supported by previous work without sacrificing performance.

Keywords: Protocol verification · Multiparty session types · Communicating state machines · Protocol fidelity · Deadlock freedom

1 Introduction

Communication protocols are key components in many safety and operation critical systems, making them prime targets for formal verification. Unfortunately, most verification problems for such protocols (e.g. deadlock freedom) are undecidable [11]. To make verification computationally tractable, several restrictions have been proposed [2, 3, 10, 14, 33, 42]. In particular, multiparty session types (MSTs) [24] have garnered a lot of attention in recent years (see, e.g., the survey by Ancona et al. [6]). In the MST setting, a protocol is specified as a global

E. Li and F. Stutz—equal contribution.

© The Author(s) 2023

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13966, pp. 350–373, 2023.

https://doi.org/10.1007/978-3-031-37709-9_17

type, which describes the desired interactions of all roles involved in the protocol. Local implementations describe behaviors for each individual role. The implementability problem for a global type asks whether there exists a collection of local implementations whose composite behavior when viewed as a communicating state machine (CSM) matches that of the global type and is deadlock-free. The synthesis problem is to compute such an implementation from an implementable global type.

MST-based approaches typically solve synthesis and implementability simultaneously via an efficient syntactic *projection operator* [18, 24, 34, 41]. Abstractly, a projection operator is a partial map from global types to collections of implementations. A projection operator proj is sound when every global type \mathbf{G} in its domain is implemented by $\text{proj}(\mathbf{G})$, and complete when every implementable global type is in its domain. Existing practical projection operators for MSTs are all incomplete (or unsound). Recently, the implementability problem was shown to be decidable for a class of MSTs via a reduction to safe realizability of globally cooperative high-level message sequence charts (HMSCs) [38]. In principle, this result yields a complete and sound projection operator for the considered class. However, this operator would not be practical. In particular, the proposed implementability check is in EXPSPACE.

Contributions. In this paper, we present the first practical sound and complete projection operator for general MSTs. The synthesis problem for implementable global types is conceptually easy [38] – the challenge lies in determining whether a global type *is* implementable. We thus separate synthesis from checking implementability. We first use a standard automata-theoretic construction to obtain a candidate implementation for a potentially non-implementable global type. However, unlike [38], we then verify the correctness of this implementation directly using efficiently checkable conditions derived from the global type. When a global type is not implementable, our constructive completeness proof provides a counterexample trace.

The resulting projection operator yields a PSPACE decision procedure for implementability. In fact, we show that the implementability problem is PSPACE-complete. These results both generalize and tighten the decidability and complexity results obtained in [38].

We evaluate a prototype of our projection algorithm on benchmarks taken from the literature. Our prototype benefits from both the efficiency of existing lightweight but incomplete syntactic projection operators [18, 24, 34, 41], and the generality of heavyweight automata-based model checking techniques [28, 36]: it handles protocols rejected by previous practical approaches while preserving the efficiency that makes MST-based techniques so attractive.

2 Motivation and Overview

Incompleteness of Existing Projection Operators. A key limitation of existing projection operators is that the implementation for each role is obtained

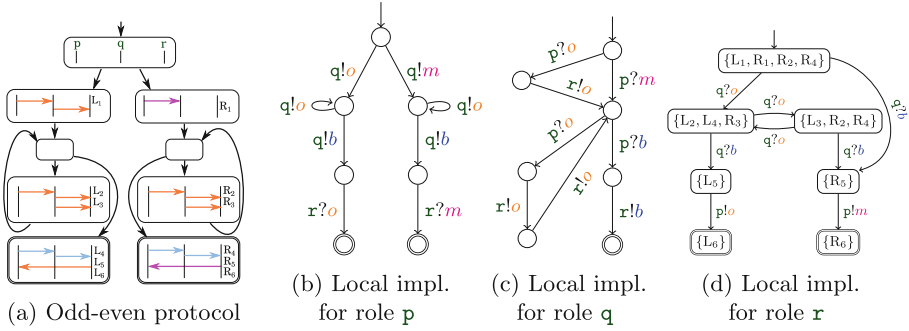


Fig. 1. Odd-even: An implementable but not (yet) projectable protocol and its local implementations

via a linear traversal of the global type, and thus shares its structure. The following example, which is not projectable by any existing approach, demonstrates how enforcing structural similarity can lead to incompleteness.

Example 2.1 (Odd-even). Consider the following global type \mathbf{G}_{oe} :

$$+ \left\{ \begin{array}{l} p \rightarrow q : o . q \rightarrow r : o . \mu t_1 . (p \rightarrow q : o . q \rightarrow r : o . q \rightarrow r : o . t_1 + p \rightarrow q : b . q \rightarrow r : b . r \rightarrow p : o . 0) \\ p \rightarrow q : m . \mu t_2 . (p \rightarrow q : o . q \rightarrow r : o . q \rightarrow r : o . t_2 + p \rightarrow q : b . q \rightarrow r : b . r \rightarrow p : m . 0) \end{array} \right.$$

A term $p \rightarrow q : m$ specifies the exchange of message m between sender p and receiver q . The term represents two local events observed separately due to asynchrony: a send event $p \triangleright q ! m$ observed by role p , and a receive event $q \triangleleft p ? m$ observed by role q . The $+$ operator denotes choice, $\mu t . G$ denotes recursion, and 0 denotes protocol termination.

Figure 1a visualizes \mathbf{G}_{oe} as an HMSC. The left and right sub-protocols respectively correspond to the top and bottom branches of the protocol. Role p chooses a branch by sending either o or m to q . On the left, q echoes this message to r . Both branches continue in the same way: p sends an arbitrary number of o messages to q , each of which is forwarded twice from q to r . Role p signals the end of the loop by sending b to q , which q forwards to r . Finally, depending on the branch, r must send o or m to p .

Figures 1b and 1c depict the structural similarity between the global type \mathbf{G}_{oe} and the implementations for p and q . For the “choicemaker” role p , the reason is evident. Role q ’s implementation collapses the continuations of both branches in the protocol into a single sub-component. For r (Fig. 1d), the situation is more complicated. Role r does not decide on or learn directly which branch is taken, but can deduce it from the parity of the number of o messages received from q : odd means left and even means right. The resulting local implementation features transitions going back and forth between the two branches that do not exist in the global type. Syntactic projection operators fail to create such transitions. ◀

One response to the brittleness of existing projection operators has been to give up on global type specifications altogether and instead revert to model checking

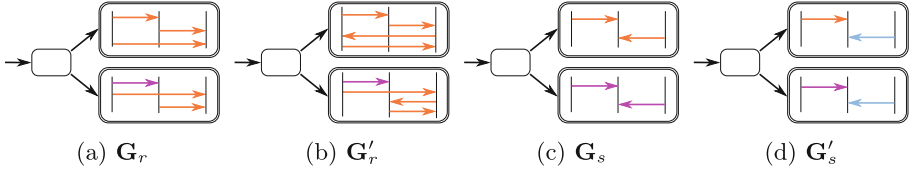


Fig. 2. High-level message sequence charts for the global types of Example 2.2.

user-provided implementations [28,36]. We posit that what needs rethinking is not the concept of global types, but rather how projections are computed and how implementability is checked.

Our Automata-Theoretic Approach. The synthesis step in our projection operator uses textbook automata-theoretic constructions. From a given global type, we derive a finite state machine, and use it to define a homomorphism automaton for each role. We then determinize this homomorphism automaton via subset construction to obtain a local candidate implementation for each role. If the global type is implementable, this construction always yields an implementation. The implementations shown in Figs. 1b to 1d are the result of applying this construction to \mathbf{G}_{oe} from Example 2.1. Notice that the state labels in Fig. 1d correspond to sets of labels in the global protocol.

Unfortunately, not all global types are implementable.

Example 2.2. Consider the following four global types also depicted in Fig. 2:

$$\begin{aligned} \mathbf{G}_r &= + \begin{cases} p \rightarrow q : o. q \rightarrow r : o. p \rightarrow r : o. 0 \\ p \rightarrow q : m. p \rightarrow r : o. q \rightarrow r : o. 0 \end{cases} & \mathbf{G}_s &= + \begin{cases} p \rightarrow q : o. r \rightarrow q : o. 0 \\ p \rightarrow q : m. r \rightarrow q : m. 0 \end{cases} \\ \mathbf{G}'_r &= + \begin{cases} p \rightarrow q : o. q \rightarrow r : o. r \rightarrow p : o. p \rightarrow r : o. 0 \\ p \rightarrow q : m. p \rightarrow r : o. r \rightarrow q : o. q \rightarrow r : o. 0 \end{cases} & \mathbf{G}'_s &= + \begin{cases} p \rightarrow q : o. r \rightarrow q : b. 0 \\ p \rightarrow q : m. r \rightarrow q : b. 0 \end{cases} \end{aligned}$$

Similar to \mathbf{G}_{oe} , in all four examples, p chooses a branch by sending either o or m to q. The global type \mathbf{G}_r is not implementable because r cannot learn which branch was chosen by p. For any local implementation of r to be able to execute both branches, it must be able to receive o from p and q in any order. Because the two send events $p \triangleright r!o$ and $q \triangleright r!o$ are independent of each other, they may be reordered. Consequently, any implementation of \mathbf{G}_r would have to permit executions that are consistent with global behaviors not described by \mathbf{G}_r , such as $p \rightarrow q : m \cdot q \rightarrow r : o \cdot p \rightarrow r : o$. Contrast this with \mathbf{G}'_r , which is implementable. In the top branch of \mathbf{G}'_r , role p can only send to r after it has received from r, which prevents the reordering of the send events $p \triangleright r!o$ and $q \triangleright r!o$. The bottom branch is symmetric. Hence, r learns p's choice based on which message it receives first.

For the global type \mathbf{G}_s , role r again cannot learn the branch chosen by p. That is, r cannot know whether to send o or m to q, leading inevitably to deadlocking executions. In contrast, \mathbf{G}'_s is again implementable because the expected behavior of r is independent of the choice by p. ◀

These examples show that the implementability question is non-trivial. To check implementability, we present conditions that precisely characterize when the subset construction for \mathbf{G} yields an implementation.

Overview. The rest of the paper is organized as follows. Section 3 contains relevant definitions for our work. Section 4 describes the synthesis step of our projection. Section 5 presents the two conditions that characterize implementability of a given global type. In Sect. 6, we prove soundness of our projection via a stronger inductive invariant guaranteeing per-role agreement on a global run of the protocol. In Sect. 7, we prove completeness by showing that our two conditions hold if a global type is implementable. In Sect. 8, we discuss the complexity of our construction and condition checks. Section 9 presents our artifact and evaluation, and Sect. 10 as well as Sect. 11 discuss related work. Additional details including omitted proofs can be found in the extended version of the paper [29].

3 Preliminaries

Words. Let Σ be a finite alphabet. Σ^* denotes the set of finite words over Σ , Σ^ω the set of infinite words, and Σ^∞ their union $\Sigma^* \cup \Sigma^\omega$. A word $u \in \Sigma^*$ is a *prefix* of word $v \in \Sigma^\infty$, denoted $u \leq v$, if there exists $w \in \Sigma^\infty$ with $u \cdot w = v$.

Message Alphabet. Let \mathcal{P} be a set of roles and \mathcal{V} be a set of messages. We define the set of *synchronous events* $\Sigma_{sync} := \{p \rightarrow q : m \mid p, q \in \mathcal{P} \text{ and } m \in \mathcal{V}\}$ where $p \rightarrow q : m$ denotes that message m is sent by p to q atomically. This is split for *asynchronous events*. For a role $p \in \mathcal{P}$, we define the alphabet $\Sigma_{p,!} = \{p \triangleright q ! m \mid q \in \mathcal{P}, m \in \mathcal{V}\}$ of *send events* and the alphabet $\Sigma_{p,?} = \{p \triangleleft q ? m \mid q \in \mathcal{P}, m \in \mathcal{V}\}$ of *receive events*. The event $p \triangleright q ! m$ denotes role p sending a message m to q , and $p \triangleleft q ? m$ denotes role p receiving a message m from q . We write $\Sigma_p = \Sigma_{p,!} \cup \Sigma_{p,?}$, $\Sigma_! = \bigcup_{p \in \mathcal{P}} \Sigma_{p,!}$, and $\Sigma_? = \bigcup_{p \in \mathcal{P}} \Sigma_{p,?}$. Finally, $\Sigma_{async} = \Sigma_! \cup \Sigma_?$. We say that p is *active* in $x \in \Sigma_{async}$ if $x \in \Sigma_p$. For each role $p \in \mathcal{P}$, we define a homomorphism \Downarrow_{Σ_p} , where $x \Downarrow_{\Sigma_p} = x$ if $x \in \Sigma_p$ and ε otherwise. We write $\mathcal{V}(w)$ to project the send and receive events in w onto their messages. We fix \mathcal{P} and \mathcal{V} in the rest of the paper.

Global Types – Syntax. Global types for MSTs [31] are defined by the grammar:

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

where p, q_i range over \mathcal{P} , m_i over \mathcal{V} , and t over a set of recursion variables.

We require each branch of a choice to be distinct: $\forall i, j \in I. i \neq j \Rightarrow (q_i, m_i) \neq (q_j, m_j)$, the sender and receiver of an atomic action to be distinct: $\forall i \in I. p \neq q_i$, and recursion to be guarded: in $\mu t . G$, there is at least one message between μt and each t in G . When $|I| = 1$, we omit \sum . For readability, we sometimes use the infix operator $+$ for choice, instead of \sum . When working with a protocol described by a global type, we write \mathbf{G} to refer to the top-level type, and we

use G to refer to its subterms. For the size of a global type, we disregard multiple occurrences of the same subterm.

We use the extended definition of global types from [31] that allows a sender to send messages to different roles in a choice. We call this *sender-driven choice*, as in [38], while it was called generalized choice in [31]. This definition subsumes classical MSTs that only allow *directed choice* [24]. The types we use focus on communication primitives and omit features like delegation or parametrization. We defer a detailed discussion of different MST frameworks to Sect. 11.

Global Types – Semantics. As a basis for the semantics of a global type \mathbf{G} , we construct a finite state machine $\mathbf{GAut}(\mathbf{G}) = (Q_{\mathbf{G}}, \Sigma_{\text{sync}}, \delta_{\mathbf{G}}, q_{0,\mathbf{G}}, F_{\mathbf{G}})$ where

- $Q_{\mathbf{G}}$ is the set of all syntactic subterms in \mathbf{G} together with the term 0,
- $\delta_{\mathbf{G}}$ is the smallest set containing $(\sum_{i \in I} \mathbf{p} \rightarrow \mathbf{q}_i : m_i . G_i, \mathbf{p} \rightarrow \mathbf{q}_i : m_i . G_i)$ for each $i \in I$, as well as $(\mu t . G', \varepsilon, G')$ and $(t, \varepsilon, \mu t . G')$ for each subterm $\mu t . G'$,
- $q_{0,\mathbf{G}} = \mathbf{G}$ and $F_{\mathbf{G}} = \{0\}$.

We define a homomorphism `split` onto the asynchronous alphabet:

$$\text{split}(\mathbf{p} \rightarrow \mathbf{q} : m) := \mathbf{p} \triangleright \mathbf{q} ! m . \mathbf{q} \triangleleft \mathbf{p} ? m .$$

The semantics $\mathcal{L}(\mathbf{G})$ of a global type \mathbf{G} is given by $\mathcal{C}^{\sim}(\text{split}(\mathcal{L}(\mathbf{GAut}(\mathbf{G}))))$ where \mathcal{C}^{\sim} is the closure under the indistinguishability relation \sim [31]. Two events are independent if they are not related by the *happened-before* relation [26]. For instance, any two send events from distinct senders are independent. Two words are indistinguishable if one can be reordered into the other by repeatedly swapping consecutive independent events. The full definition is in the extended version [29].

Communicating State Machine [11]. $\mathcal{A} = \{\{A_{\mathbf{p}}\}_{\mathbf{p} \in \mathcal{P}}$ is a CSM over \mathcal{P} and \mathcal{V} if $A_{\mathbf{p}}$ is a finite state machine over $\Sigma_{\mathbf{p}}$ for every $\mathbf{p} \in \mathcal{P}$, denoted by $(Q_{\mathbf{p}}, \Sigma_{\mathbf{p}}, \delta_{\mathbf{p}}, q_{0,\mathbf{p}}, F_{\mathbf{p}})$. Let $\prod_{\mathbf{p} \in \mathcal{P}} s_{\mathbf{p}}$ denote the set of global states and $\text{Chan} = \{(\mathbf{p}, \mathbf{q}) \mid \mathbf{p}, \mathbf{q} \in \mathcal{P}, \mathbf{p} \neq \mathbf{q}\}$ denote the set of channels. A *configuration* of \mathcal{A} is a pair (\vec{s}, ξ) , where \vec{s} is a global state and $\xi : \text{Chan} \rightarrow \mathcal{V}^*$ is a mapping from each channel to a sequence of messages. We use $\vec{s}_{\mathbf{p}}$ to denote the state of \mathbf{p} in \vec{s} . The CSM transition relation, denoted \rightarrow , is defined as follows.

- $(\vec{s}, \xi) \xrightarrow{\mathbf{p} \triangleright \mathbf{q} ! m} (\vec{s}', \xi')$ if $(\vec{s}_{\mathbf{p}}, \mathbf{p} \triangleright \mathbf{q} ! m, \vec{s}'_{\mathbf{p}}) \in \delta_{\mathbf{p}}$, $\vec{s}_{\mathbf{r}} = \vec{s}'_{\mathbf{r}}$ for every role $\mathbf{r} \neq \mathbf{p}$, $\xi'(\mathbf{p}, \mathbf{q}) = \xi(\mathbf{p}, \mathbf{q}) \cdot m$ and $\xi'(c) = \xi(c)$ for every other channel $c \in \text{Chan}$.
- $(\vec{s}, \xi) \xrightarrow{\mathbf{q} \triangleleft \mathbf{p} ? m} (\vec{s}', \xi')$ if $(\vec{s}_{\mathbf{q}}, \mathbf{q} \triangleleft \mathbf{p} ? m, \vec{s}'_{\mathbf{q}}) \in \delta_{\mathbf{q}}$, $\vec{s}_{\mathbf{r}} = \vec{s}'_{\mathbf{r}}$ for every role $\mathbf{r} \neq \mathbf{q}$, $\xi(\mathbf{p}, \mathbf{q}) = m \cdot \xi'(\mathbf{p}, \mathbf{q})$ and $\xi'(c) = \xi(c)$ for every other channel $c \in \text{Chan}$.

In the initial configuration (\vec{s}_0, ξ_0) , each role's state in \vec{s}_0 is the initial state $q_{0,\mathbf{p}}$ of $A_{\mathbf{p}}$, and ξ_0 maps each channel to ε . A configuration (\vec{s}, ξ) is said to be *final* iff $\vec{s}_{\mathbf{p}}$ is final for every \mathbf{p} and ξ maps each channel to ε . Runs and traces are defined in the expected way. A run is *maximal* if either it is finite and ends in a final

configuration, or it is infinite. The language $\mathcal{L}(\mathcal{A})$ of the CSM \mathcal{A} is defined as the set of maximal traces. A configuration (\vec{s}, ξ) is a *deadlock* if it is not final and has no outgoing transitions. A CSM is *deadlock-free* if no reachable configuration is a deadlock.

Finally, implementability is formalized as follows.

Definition 3.1 (Implementability [31]). *A global type \mathbf{G} is implementable if there exists a CSM $\{\{A_p\}_{p \in \mathcal{P}}\}$ such that the following two properties hold: (i) protocol fidelity: $\mathcal{L}(\{\{A_p\}_{p \in \mathcal{P}}\}) = \mathcal{L}(\mathbf{G})$, and (ii) deadlock freedom: $\{\{A_p\}_{p \in \mathcal{P}}\}$ is deadlock-free. We say that $\{\{A_p\}_{p \in \mathcal{P}}\}$ implements \mathbf{G} .*

4 Synthesizing Implementations

The construction is carried out in two steps. First, for each role $p \in \mathcal{P}$, we define an intermediate state machine $\mathbf{GAut}(\mathbf{G})\downarrow_p$ that is a homomorphism of $\mathbf{GAut}(\mathbf{G})$. We call $\mathbf{GAut}(\mathbf{G})\downarrow_p$ the *projection by erasure* for p , defined below.

Definition 4.1 (Projection by Erasure). *Let \mathbf{G} be some global type with its state machine $\mathbf{GAut}(\mathbf{G}) = (Q_{\mathbf{G}}, \Sigma_{sync}, \delta_{\mathbf{G}}, q_{0,\mathbf{G}}, F_{\mathbf{G}})$. For each role $p \in \mathcal{P}$, we define the state machine $\mathbf{GAut}(\mathbf{G})\downarrow_p = (Q_{\mathbf{G}}, \Sigma_p \uplus \{\varepsilon\}, \delta_{\downarrow}, q_{0,\mathbf{G}}, F_{\mathbf{G}})$ where $\delta_{\downarrow} := \{q \xrightarrow{split(a)\downarrow_{\Sigma_p}} q' \mid q \xrightarrow{a} q' \in \delta_{\mathbf{G}}\}$. By definition of $split(-)$, it holds that $split(a)\downarrow_{\Sigma_p} \in \Sigma_p \uplus \{\varepsilon\}$.*

Then, we determinize $\mathbf{GAut}(\mathbf{G})\downarrow_p$ via a standard subset construction to obtain a deterministic local state machine for p .

Definition 4.2 (Subset Construction). *Let \mathbf{G} be a global type and p be a role. Then, the subset construction for p is defined as*

$$\mathcal{C}(\mathbf{G}, p) = (Q_p, \Sigma_p, \delta_p, s_{0,p}, F_p) \text{ where}$$

- $\delta(s, a) := \{q' \in Q_{\mathbf{G}} \mid \exists q \in s, q \xrightarrow{a} \varepsilon^* q' \in \delta_{\downarrow}\}$, for every $s \subseteq Q_{\mathbf{G}}$ and $a \in \Sigma_p$
- $s_{0,p} := \{q \in Q_{\mathbf{G}} \mid q_{0,\mathbf{G}} \xrightarrow{\varepsilon^*} q \in \delta_{\downarrow}\}$,
- $Q_p := \text{lfp}_{\{s_{0,p}\}}^{\subseteq} \lambda Q. Q \cup \{\delta(s, a) \mid s \in Q \wedge a \in \Sigma_p\} \setminus \{\emptyset\}$, and
- $\delta_p := \delta|_{Q_p \times \Sigma_p}$
- $F_p := \{s \in Q_p \mid s \cap F_{\mathbf{G}} \neq \emptyset\}$

Note that the construction ensures that Q_p only contains subsets of $Q_{\mathbf{G}}$ whose states are reachable via the same traces, i.e. we typically have $|Q_p| \ll 2^{|Q_{\mathbf{G}}|}$.

The following characterization is immediate from the subset construction; the proof can be found in the extended version [29].

Lemma 4.3. *Let \mathbf{G} be a global type, r be a role, and $\mathcal{C}(\mathbf{G}, r)$ be its subset construction. If w is a trace of $\mathbf{GAut}(\mathbf{G})$, $split(w)\downarrow_{\Sigma_r}$ is a trace of $\mathcal{C}(\mathbf{G}, r)$. If u is a trace of $\mathcal{C}(\mathbf{G}, r)$, there is a trace w of $\mathbf{GAut}(\mathbf{G})$ such that $split(w)\downarrow_{\Sigma_r} = u$. It holds that $\mathcal{L}(\mathbf{G})\downarrow_{\Sigma_r} = \mathcal{L}(\mathcal{C}(\mathbf{G}, r))$.*

Using this lemma, we show that the CSM $\{\{\mathcal{C}(\mathbf{G}, \mathbf{p})\}\}_{\mathbf{p} \in \mathcal{P}}$ preserves all behaviors of \mathbf{G} .

Lemma 4.4. *For all global types \mathbf{G} , $\mathcal{L}(\mathbf{G}) \subseteq \mathcal{L}(\{\{\mathcal{C}(\mathbf{G}, \mathbf{p})\}\}_{\mathbf{p} \in \mathcal{P}})$.*

We briefly sketch the proof here. Given that $\{\{\mathcal{C}(\mathbf{G}, \mathbf{p})\}\}_{\mathbf{p} \in \mathcal{P}}$ is deterministic, to prove language inclusion it suffices to prove the inclusion of the respective prefix sets:

$$\text{pref}(\mathcal{L}(\mathbf{G})) \subseteq \text{pref}(\mathcal{L}(\{\{\mathcal{C}(\mathbf{G}, \mathbf{p})\}\}_{\mathbf{p} \in \mathcal{P}}))$$

Let w be a word in $\mathcal{L}(\mathbf{G})$. If w is finite, membership in $\mathcal{L}(\{\{\mathcal{C}(\mathbf{G}, \mathbf{p})\}\}_{\mathbf{p} \in \mathcal{P}})$ is immediate from the claim above. If w is infinite, we show that w has an infinite run in $\{\{\mathcal{C}(\mathbf{G}, \mathbf{p})\}\}_{\mathbf{p} \in \mathcal{P}}$ using König's Lemma. We construct an infinite graph $\mathcal{G}_w(V, E)$ with $V := \{v_\rho \mid \text{trace}(\rho) \leq w\}$ and $E := \{(v_{\rho_1}, v_{\rho_2}) \mid \exists x \in \Sigma_{\text{asyn.c.}} \text{trace}(\rho_2) = \text{trace}(\rho_1) \cdot x\}$. Because $\{\{\mathcal{C}(\mathbf{G}, \mathbf{p})\}\}_{\mathbf{p} \in \mathcal{P}}$ is deterministic, \mathcal{G}_w is a tree rooted at v_ε , the vertex corresponding to the empty run. By König's Lemma, every infinite tree contains either a vertex of infinite degree or an infinite path. Because $\{\{\mathcal{C}(\mathbf{G}, \mathbf{p})\}\}_{\mathbf{p} \in \mathcal{P}}$ consists of a finite number of communicating state machines, the last configuration of any run has a finite number of next configurations, and \mathcal{G}_w is finitely branching. Therefore, there must exist an infinite path in \mathcal{G}_w representing an infinite run for w , and thus $w \in \mathcal{L}(\{\{\mathcal{C}(\mathbf{G}, \mathbf{p})\}\}_{\mathbf{p} \in \mathcal{P}})$.

The proof of the inclusion of prefix sets proceeds by structural induction and primarily relies on Lemma 4.3 and the fact that all prefixes in $\mathcal{L}(\mathbf{G})$ respect the order of send before receive events.

5 Checking Implementability

We now turn our attention to checking implementability of a CSM produced by the subset construction. We revisit the global types from Example 2.2 (also shown in Fig. 2), which demonstrate that the naive subset construction does not always yield a sound implementation. From these examples, we distill our conditions that precisely identify the implementable global types.

In general, a global type \mathbf{G} is not implementable when the agreement on a global run of $\text{GAut}(\mathbf{G})$ among all participating roles cannot be conveyed via sending and receiving messages alone. When this happens, roles can take locally permitted transitions that commit to incompatible global runs, resulting in a trace that is not specified by \mathbf{G} . Consequently, our conditions need to ensure that when a role \mathbf{p} takes a transition in $\mathcal{C}(\mathbf{G}, \mathbf{p})$, it only commits to global runs that are consistent with the local views of all other roles. We discuss the relevant conditions imposed on send and receive transitions separately.

Send Validity. Consider \mathbf{G}_s from Example 2.2. The CSM $\{\{\mathcal{C}(\mathbf{G}_s, \mathbf{p})\}\}_{\mathbf{p} \in \mathcal{P}}$ has an execution with the trace $\mathbf{p} \triangleright \mathbf{q}!o \cdot \mathbf{q} \triangleleft \mathbf{p} ? o \cdot \mathbf{r} \triangleright \mathbf{q}!m$. This trace is possible because the initial state of $\mathcal{C}(\mathbf{G}_s, \mathbf{r})$, $s_{0,\mathbf{r}}$, contains two states of $\text{GAut}(\mathbf{G}_s) \downarrow_{\mathbf{r}}$, each of which has a single outgoing send transition labeled with $\mathbf{r} \triangleright \mathbf{q}!o$ and $\mathbf{r} \triangleright \mathbf{q}!m$ respectively. Both of these transitions are always enabled in $s_{0,\mathbf{r}}$, meaning that \mathbf{r} can send

$r \triangleright q!m$ even when p has chosen the top branch and q expects to receive o instead of m from r . This results in a deadlock. In contrast, while the state $s_{0,r}$ in $\mathcal{C}(\mathbf{G}'_s, r)$ likewise contains two states of $\mathbf{GAut}(\mathbf{G}'_s) \downarrow_r$, each with a single outgoing send transition, now both transitions are labeled with $r \triangleright q!b$. These two transitions collapse to a single one in $\mathcal{C}(\mathbf{G}'_s, r)$. This transition is consistent with both possible local views that p and q might hold on the global run.

Intuitively, to prevent the emergence of inconsistent local views from send transitions of $\mathcal{C}(\mathbf{G}, p)$, we must enforce that for every state $s \in Q_p$ with an outgoing send transition labeled x , a transition labeled x must be enabled in all states of $\mathbf{GAut}(\mathbf{G}) \downarrow_p$ represented by s . We use the following auxiliary definition to formalize this intuition subsequently.

Definition 5.1 (Transition Origin and Destination). *Let $s \xrightarrow{x} s' \in \delta_p$ be a transition in $\mathcal{C}(\mathbf{G}, p)$ and δ_\downarrow be the transition relation of $\mathbf{GAut}(\mathbf{G}) \downarrow_p$. We define the set of transition origins $\text{tr-orig}(s \xrightarrow{x} s')$ and transition destinations $\text{tr-dest}(s \xrightarrow{x} s')$ as follows:*

$$\begin{aligned} \text{tr-orig}(s \xrightarrow{x} s') &:= \{G \in s \mid \exists G' \in s'. G \xrightarrow{x^*} G' \in \delta_\downarrow\} \text{ and} \\ \text{tr-dest}(s \xrightarrow{x} s') &:= \{G' \in s' \mid \exists G \in s. G \xrightarrow{x^*} G' \in \delta_\downarrow\} . \end{aligned}$$

Our condition on send transitions is then stated below.

Definition 5.2 (Send Validity). *$\mathcal{C}(\mathbf{G}, p)$ satisfies Send Validity iff every send transition $s \xrightarrow{x} s' \in \delta_p$ is enabled in all states contained in s :*

$$\forall s \xrightarrow{x} s' \in \delta_p. x \in \Sigma_{p,!} \implies \text{tr-orig}(s \xrightarrow{x} s') = s .$$

Receive Validity. To motivate our condition on receive transitions, let us revisit \mathbf{G}_r from Example 2.2. The CSM $\{\{\mathcal{C}(\mathbf{G}_r, p)\}\}_{p \in \mathcal{P}}$ recognizes the following trace not in the global type language $\mathcal{L}(\mathbf{G}_r)$:

$$p \triangleright q!o \cdot q \triangleleft p?o \cdot q \triangleright r!o \cdot p \triangleright r!o \cdot r \triangleleft p?o \cdot r \triangleleft q?o .$$

The issue lies with r which cannot distinguish between the two branches in \mathbf{G}_r . The initial state $s_{0,r}$ of $\mathcal{C}(\mathbf{G}_r, r)$ has two states of $\mathbf{GAut}(\mathbf{G}_r)$ corresponding to the subterms $G_t := q \rightarrow r : o . p \rightarrow r : o . 0$ and $G_b := p \rightarrow r : o . q \rightarrow r : o . 0$. Here, G_t and G_b are the top and bottom branch of \mathbf{G}_r respectively. This means that there are outgoing transitions in $s_{0,r}$ labeled with $r \triangleleft p?o$ and $r \triangleleft q?o$. If r takes the transition labeled $r \triangleleft p?o$, it commits to the bottom branch G_b . However, observe that the message o from p can also be available at this time point if the other roles follow the top branch G_t . This is because p can send o to r without waiting for r to first receive from q . In this scenario, the roles disagree on which global run of $\mathbf{GAut}(\mathbf{G}_r)$ to follow, resulting in the violating trace above.

Contrast this with \mathbf{G}'_r . Here, $s_{0,r}$ again has outgoing transitions labeled with $r \triangleleft p?o$ and $r \triangleleft q?o$. However, if r takes the transition labeled $r \triangleleft p?o$, committing to the bottom branch, no disagreement occurs. This is because if the other roles

are following the top branch, then p is blocked from sending to r until after it has received confirmation that r has received its first message from q .

For a receive transition $s \xrightarrow{x} s_1$ in $\mathcal{C}(\mathbf{G}, p)$ to be safe, we must enforce that the receive event x cannot also be available due to reordered sent messages in the continuation $G_2 \in s_2$ of another outgoing receive transition $s \xrightarrow{y} s_2$. To formalize this condition, we use the set $M_{(G\dots)}^{\mathcal{B}}$ of *available messages* for a syntactic subterm G of \mathbf{G} and a set of *blocked* roles \mathcal{B} . This notion was already defined in [31, Sec. 2.2]. Intuitively, $M_{(G\dots)}^{\mathcal{B}}$ consists of all send events $q \triangleright r!m$ that can occur on the traces of G such that m will be the first message added to channel (q, r) before any of the roles in \mathcal{B} takes a step.

Available Messages. The set of available messages is recursively defined on the structure of the global type. To obtain all possible messages, we need to unfold the distinct recursion variables once. For this, we define a map $get\mu$ from variable to subterms and write $get\mu_{\mathbf{G}}$ for $get\mu(\mathbf{G})$:

$$\begin{aligned} get\mu(0) &:= [] & get\mu(t) &:= [] & get\mu(\mu t.G) &:= [t \mapsto G] \cup get\mu(G) \\ get\mu(\sum_{i \in I} p \rightarrow q_i : m_i . G_i) &:= \bigcup_{i \in I} get\mu(G_i) \end{aligned}$$

The function $M_{(-\dots)}^{\mathcal{B}, T}$ keeps a set of unfolded variables T , which is empty initially.

$$M_{(0\dots)}^{\mathcal{B}, T} := \emptyset \quad M_{(\mu t.G\dots)}^{\mathcal{B}, T} := M_{(G\dots)}^{\mathcal{B}, T \cup \{t\}} \quad M_{(t\dots)}^{\mathcal{B}, T} := \begin{cases} \emptyset & \text{if } t \in T \\ M_{(get\mu_{\mathbf{G}}(t)\dots)}^{\mathcal{B}, T \cup \{t\}} & \text{if } t \notin T \end{cases}$$

$$M_{(\sum_{i \in I} p \rightarrow q_i : m_i . G_i\dots)}^{\mathcal{B}, T} := \begin{cases} \bigcup_{i \in I, m \in \mathcal{V}(M_{(G_i\dots)}^{\mathcal{B}, T} \setminus \{q_i \triangleleft p?m\})} \cup \{q_i \triangleleft p?m_i\} & \text{if } p \notin \mathcal{B} \\ \bigcup_{i \in I} M_{(G_i\dots)}^{\mathcal{B} \cup \{q_i\}, T} & \text{if } p \in \mathcal{B} \end{cases}$$

We write $M_{(G\dots)}^{\mathcal{B}}$ for $M_{(G\dots)}^{\mathcal{B}, \emptyset}$. If \mathcal{B} is a singleton set, we omit set notation and write $M_{(G\dots)}^p$ for $M_{(G\dots)}^{\{p\}}$. The set of available messages captures the possible states of all channels before a given receive transition is taken.

Definition 5.3 (Receive Validity). $\mathcal{C}(\mathbf{G}, p)$ satisfies Receive Validity iff no receive transition is enabled in an alternative continuation that originates from the same source state:

$$\begin{aligned} \forall s \xrightarrow{p \triangleleft q_1 ? m_1} s_1, s \xrightarrow{p \triangleleft q_2 ? m_2} s_2 \in \delta_p. \\ q_1 \neq q_2 \implies \forall G_2 \in \text{tr-dest}(s \xrightarrow{p \triangleleft q_2 ? m_2} s_2). q_1 \triangleright p!m_1 \notin M_{(G_2\dots)}^p. \end{aligned}$$

Subset Projection. We are now ready to define our projection operator.

Definition 5.4 (Subset Projection of \mathbf{G}). The subset projection $\mathcal{P}(\mathbf{G}, p)$ of \mathbf{G} onto p is $\mathcal{C}(\mathbf{G}, p)$ if it satisfies Send Validity and Receive Validity. We lift this operation to a partial function from global types to CSMs in the expected way.

We conclude our discussion with an observation about the syntactic structure of the subset projection: Send Validity implies that no state has both outgoing send and receive transitions (also known as mixed choice).

Corollary 5.5 (No Mixed Choice). If $\mathcal{P}(\mathbf{G}, p)$ satisfies Send Validity, then for all $s \xrightarrow{x_1} s_1, s \xrightarrow{x_2} s_2 \in \delta_p, x_1 \in \Sigma!, \text{ iff } x_2 \in \Sigma!$.

6 Soundness

In this section, we prove the soundness of our subset projection, stated as follows.

Theorem 6.1. *Let \mathbf{G} be a global type and $\{\{\mathcal{P}(\mathbf{G}, \mathfrak{p})\}\}_{\mathfrak{p} \in \mathcal{P}}$ be the subset projection. Then, $\{\{\mathcal{P}(\mathbf{G}, \mathfrak{p})\}\}_{\mathfrak{p} \in \mathcal{P}}$ implements \mathbf{G} .*

Recall that implementability is defined as protocol fidelity and deadlock freedom. Protocol fidelity consists of two language inclusions. The first inclusion, $\mathcal{L}(\mathbf{G}) \subseteq \mathcal{L}(\{\{\mathcal{P}(\mathbf{G}, \mathfrak{p})\}\}_{\mathfrak{p} \in \mathcal{P}})$, enforces that the subset projection generates at least all behaviors of the global type. We showed in Lemma 4.4 that this holds for the subset construction alone (without Send and Receive Validity).

The second inclusion, $\mathcal{L}(\{\{\mathcal{P}(\mathbf{G}, \mathfrak{p})\}\}_{\mathfrak{p} \in \mathcal{P}}) \subseteq \mathcal{L}(\mathbf{G})$, enforces that no new behaviors are introduced. The proof of this direction relies on a stronger inductive invariant that we show for all traces of the subset projection. As discussed in Sect. 5, violations of implementability occur when roles commit to global runs that are inconsistent with the local views of other roles. Our inductive invariant states the exact opposite: that all local views are consistent with one another. First, we formalize the local view of a role.

Definition 6.2 (Possible run sets). *Let \mathbf{G} be a global type and $\text{GAut}(\mathbf{G})$ be the corresponding state machine. Let \mathfrak{p} be a role and $w \in \Sigma_{\text{async}}^*$ be a word. We define the set of possible runs $R_{\mathfrak{p}}^{\mathbf{G}}(w)$ as all maximal runs of $\text{GAut}(\mathbf{G})$ that are consistent with \mathfrak{p} 's local view of w :*

$$R_{\mathfrak{p}}^{\mathbf{G}}(w) := \{ \rho \text{ is a maximal run of } \text{GAut}(\mathbf{G}) \mid w \downarrow_{\Sigma_{\mathfrak{p}}} \leq \text{split}(\text{trace}(\rho)) \downarrow_{\Sigma_{\mathfrak{p}}} \} .$$

While Definition 6.2 captures the set of maximal runs that are consistent with the local view of a single role, we would like to refer to the set of runs that is consistent with the local view of all roles. We formalize this as the intersection of the possible run sets for all roles, which we denote as

$$I(w) := \bigcap_{\mathfrak{p} \in \mathcal{P}} R_{\mathfrak{p}}^{\mathbf{G}}(w) .$$

With these definitions in hand, we can now formulate our inductive invariant:

Lemma 6.3. *Let \mathbf{G} be a global type and $\{\{\mathcal{P}(\mathbf{G}, \mathfrak{p})\}\}_{\mathfrak{p} \in \mathcal{P}}$ be the subset projection. Let w be a trace of $\{\{\mathcal{P}(\mathbf{G}, \mathfrak{p})\}\}_{\mathfrak{p} \in \mathcal{P}}$. It holds that $I(w)$ is non-empty.*

The reasoning for the sufficiency of Lemma 6.3 is included in the proof of Theorem 6.1, found in the extended version [29]. In the rest of this section, we focus our efforts on how to show this inductive invariant, namely that the intersection of all roles' possible run sets is non-empty.

We begin with the observation that the empty trace ε is consistent with all runs. As a result, $I(\varepsilon) = \bigcap_{\mathfrak{p} \in \mathcal{P}} R_{\mathfrak{p}}^{\mathbf{G}}(\varepsilon)$ contains all maximal runs in $\text{GAut}(\mathbf{G})$. By definition, state machines for global types include at least one run, and the base case is trivially discharged. Intuitively, $I(w)$ shrinks as more events are appended

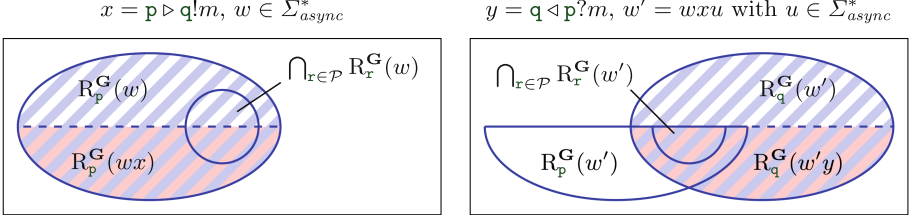


Fig. 3. Evolution of $R^G(-)$ sets when p sends a message m and q receives it.

to w , but we show that at no point does it shrink to \emptyset . We consider the cases where a send or receive event is appended to the trace separately, and show that the intersection set shrinks in a principled way that preserves non-emptiness. In fact, when a trace is extended with a receive event, Receive Validity guarantees that the intersection set does not shrink at all.

Lemma 6.4. *Let \mathbf{G} be a global type and $\{\{\mathcal{P}(\mathbf{G}, p)\}\}_{p \in \mathcal{P}}$ be the subset projection. Let wx be a trace of $\{\{\mathcal{P}(\mathbf{G}, p)\}\}_{p \in \mathcal{P}}$ such that $x \in \Sigma_{\tau}$. Then, $I(w) = I(wx)$.*

To prove this equality, we further refine our characterization of intersection sets. In particular, we show that in the receive case, the intersection between the sender and receiver's possible run sets stays the same, i.e.

$$R_p^{\mathbf{G}}(w) \cap R_q^{\mathbf{G}}(w) = R_p^{\mathbf{G}}(wx) \cap R_q^{\mathbf{G}}(wx) .$$

Note that it is not the case that the receiver only follows a subset of the sender's possible runs. In other words, $R_q^{\mathbf{G}}(w) \subseteq R_p^{\mathbf{G}}(w)$ is not inductive. The equality above simply states that a receive action can only eliminate runs that have already been eliminated by its sender. Figure 3 depicts this relation.

Given that the intersection set strictly shrinks, the burden of eliminating runs must then fall upon send events. We show that send transitions shrink the possible run set of the sender in a way that is *prefix-preserving*. To make this more precise, we introduce the following definition on runs.

Definition 6.5 (Unique splitting of a possible run). *Let \mathbf{G} be a global type, p a role, and $w \in \Sigma_{async}^*$ a word. Let ρ be a possible run in $R_p^{\mathbf{G}}(w)$. We define the longest prefix of ρ matching w :*

$$\alpha' := \max\{\rho' \mid \rho' \leq \rho \wedge \mathbf{split}(\mathbf{trace}(\rho')) \Downarrow_{\Sigma_p} \leq w \Downarrow_{\Sigma_p}\} .$$

If $\alpha' \neq \rho$, we can split ρ into $\rho = \alpha \cdot G \xrightarrow{l} G' \cdot \beta$ where $\alpha' = \alpha \cdot G$, G' denotes the state following G , and β denotes the suffix of ρ following $\alpha \cdot G \cdot G'$. We call $\alpha \cdot G \xrightarrow{l} G' \cdot \beta$ the unique splitting of ρ for p matching w . We omit the role p when obvious from context. This splitting is always unique because the maximal prefix of any $\rho \in R_p^{\mathbf{G}}(w)$ matching w is unique.

When role p fires a send transition $p \triangleright q!m$, any run $\rho = \alpha \cdot G \xrightarrow{l} G' \cdot \beta$ in p 's possible run with $\text{split}(l) \downarrow_{\Sigma_p} \neq p \triangleright q!m$ is eliminated. While the resulting possible run set could no longer contain runs that end with $G' \cdot \beta$, Send Validity guarantees that it must contain runs that begin with $\alpha \cdot G$. This is formalized by the following lemma.

Lemma 6.6. *Let \mathbf{G} be a global type and $\{\{\mathcal{P}(\mathbf{G}, p)\}\}_{p \in \mathcal{P}}$ be the subset projection. Let wx be a trace of $\{\{\mathcal{P}(\mathbf{G}, p)\}\}_{p \in \mathcal{P}}$ such that $x \in \Sigma_1 \cap \Sigma_p$ for some $p \in \mathcal{P}$. Let ρ be a run in $I(w)$, and $\alpha \cdot G \xrightarrow{l} G' \cdot \beta$ be the unique splitting of ρ for p with respect to w . Then, there exists a run ρ' in $I(wx)$ such that $\alpha \cdot G \leq \rho'$.*

This concludes our discussion of the send and receive cases in the inductive step to show the non-emptiness of the intersection of all roles' possible run sets. The full proofs and additional definitions can be found in the extended version [29].

7 Completeness

In this section, we prove completeness of our approach. While soundness states that if a global type's subset projection is defined, it then implements the global type, completeness considers the reverse direction.

Theorem 7.1 (Completeness). *If \mathbf{G} is implementable, then $\{\{\mathcal{P}(\mathbf{G}, p)\}\}_{p \in \mathcal{P}}$ is defined.*

We sketch the proof and refer to the extended version [29] for the full proof.

From the assumption that \mathbf{G} is implementable, we know there exists a witness CSM that implements \mathbf{G} . While the soundness proof picks our subset projection as the existential witness for showing implementability – thereby allowing us to reason directly about a particular implementation – completeness only guarantees the existence of some witness CSM. We cannot assume without loss of generality that this witness CSM is our subset construction; however, we must use the fact that it implements \mathbf{G} to show that Send and Receive Validity hold on our subset construction.

We proceed via proof by contradiction: we assume the negation of Send and Receive Validity for the subset construction, and show a contradiction to the fact that this witness CSM implements \mathbf{G} . In particular, we contradict protocol fidelity (Definition 3.1(i)), stating that the witness CSM generates precisely the language $\mathcal{L}(\mathbf{G})$. To do so, we exploit a simulation argument: we first show that the negation of Send and Receive Validity forces the subset construction to recognize a trace that is not a prefix of any word in $\mathcal{L}(\mathbf{G})$. Then, we show that this trace must also be recognized by the witness CSM, under the assumption that the witness CSM implements \mathbf{G} .

To highlight the constructive nature of our proof, we convert our proof obligation to a witness construction obligation. To contradict protocol fidelity, it suffices to construct a witness trace v_0 satisfying two properties, where $\{\{B_p\}\}_{p \in \mathcal{P}}$ is our witness CSM:

- (a) v_0 is a trace of $\{\{B_p\}_{p \in \mathcal{P}}\}$, and
 (b) the run intersection set of v_0 is empty: $I(v_0) = \bigcap_{p \in \mathcal{P}} R_p^{\mathbf{G}}(v_0) = \emptyset$.

We first establish the sufficiency of conditions (a) and (b). Because $\{\{B_p\}_{p \in \mathcal{P}}\}$ is deadlock-free by assumption, every prefix extends to a maximal trace. Thus, to prove the inequality of the two languages $\mathcal{L}(\{\{B_p\}_{p \in \mathcal{P}}\})$ and $\mathcal{L}(\mathbf{G})$, it suffices to prove the inequality of their respective prefix sets. In turn, it suffices to show the existence of a prefix of a word in one language that is not a prefix of any word in the other. We choose to construct a prefix in the CSM language that is not a prefix in $\mathcal{L}(\mathbf{G})$. We again leverage the definition of intersection sets (Definition 6.2) to weaken the property of language non-membership to the property of having an empty intersection set as follows. By the semantics of $\mathcal{L}(\mathbf{G})$, for any $w \in \mathcal{L}(\mathbf{G})$, there exists $w' \in \text{split}(\mathcal{L}(\mathbf{GAut}(\mathbf{G})))$ with $w \sim w'$. For any $w' \in \text{split}(\mathcal{L}(\mathbf{GAut}(\mathbf{G})))$, it trivially holds that w' has a non-empty intersection set. Because intersection sets are invariant under the indistinguishability relation \sim , w must also have a non-empty intersection set. Since intersection sets are monotonically decreasing, if the intersection set of w is non-empty, then for any $v \leq w$, the intersection set of v is also non-empty. Modus tollens of the chain of reasoning above tells us that in order to show a word is not a prefix in $\mathcal{L}(\mathbf{G})$, it suffices to show that its intersection set is empty.

Having established the sufficiency of properties (a) and (b) for our witness construction, we present the steps to construct v_0 from the negation of Send and Receive Validity respectively. We start by constructing a trace in $\{\{\mathcal{C}(\mathbf{G}, p)_p\}_{p \in \mathcal{P}}\}$ that satisfies (b), and then show that $\{\{B_p\}_{p \in \mathcal{P}}\}$ also recognizes the trace, thereby satisfying (a). In both cases, let p be the role and s be the state for which the respective validity condition is violated.

Send Validity (Definition 5.2). Let $s \xrightarrow{p \triangleright q!m} s' \in \delta_p$ be a transition such that

$$\text{tr-orig}(s \xrightarrow{p \triangleright q!m} s') \neq s .$$

First, we find a trace u of $\{\{\mathcal{C}(\mathbf{G}, p)_p\}_{p \in \mathcal{P}}\}$ that satisfies: (1) role p is in state s in the CSM configuration reached via u , and (2) the run of $\mathbf{GAut}(\mathbf{G})$ on u visits a state in $s \setminus \text{tr-orig}(s \xrightarrow{p \triangleright q!m} s')$. We obtain such a witness u from the $\text{split}(\text{trace}(-))$ of a run prefix of $\mathbf{GAut}(\mathbf{G})$ that ends in some state in $s \setminus \text{tr-orig}(s \xrightarrow{p \triangleright q!m} s')$. Any prefix thus obtained satisfies (1) by definition of $\mathcal{C}(\mathbf{G}, p)$, and satisfies (2) by construction. Due to the fact that send transitions are always enabled in a CSM, $u \cdot p \triangleright q!m$ must also be a trace of $\{\{\mathcal{C}(\mathbf{G}, p)_p\}_{p \in \mathcal{P}}\}$, thus satisfying property (a) by a simulation argument. We then argue that $u \cdot p \triangleright q!m$ satisfies property (b), stating that $I(u \cdot p \triangleright q!m)$ is empty: the negation of Send Validity gives that there exist no run extensions from our candidate state in $s \setminus \text{tr-orig}(s \xrightarrow{p \triangleright q!m} s')$ with the immediate next action $p \rightarrow q : m$, and therefore there exists no maximal run in $\mathbf{GAut}(\mathbf{G})$ consistent with $u \cdot p \triangleright q!m$.

Receive Validity (Definition 5.3). Let $s \xrightarrow{p \triangleleft q_1 ? m_1} s_1$ and $s \xrightarrow{p \triangleleft q_2 ? m_2} s_2 \in \delta_p$ be two transitions, and let $G_2 \in \text{tr-dest}(s \xrightarrow{p \triangleleft q_2 ? m_2} s_2)$ such that

$$q_1 \neq q_2 \text{ and } q_1 \triangleright p!m_1 \in M_{(G_2\dots)}^P .$$

Constructing the witness v_0 pivots on finding a trace u of $\{\{\mathcal{C}(\mathbf{G}, p)\}\}_{p \in \mathcal{P}}$ such that both $u \cdot p \triangleleft q_1 ? m_1$ and $u \cdot p \triangleleft q_2 ? m_2$ are traces of $\{\{\mathcal{C}(\mathbf{G}, p)\}\}_{p \in \mathcal{P}}$. Equivalently, we show there exists a reachable configuration of $\{\{\mathcal{C}(\mathbf{G}, p)\}\}_{p \in \mathcal{P}}$ in which p can receive either message from distinct senders q_1 and q_2 . Formally, the local state of p has two outgoing states labeled with $p \triangleleft q_1 ? m_1$ and $p \triangleleft q_2 ? m_2$, and the channels q_1, p and q_2, p have m_1 and m_2 at their respective heads. We construct such a u by considering a run in $\mathbf{GAut}(\mathbf{G})$ that contains two transitions labeled with $q_1 \rightarrow p : m_1$ and $q_2 \rightarrow p : m_2$. Such a run must exist due to the negation of Receive Validity. We start with the split trace of this run, and argue that, from the definition of $M(-)$ and the indistinguishability relation \sim , we can perform iterative reorderings using \sim to bubble the send action $q_1 \triangleright p!m_1$ to the position before the receive action $p \triangleleft q_2 ? m_2$. Then, (a) for $u \cdot p \triangleleft q_1 ? m_1$ holds by a simulation argument. We then separately show that (b) holds for $p \triangleleft q_1 ? m_1$ using similar reasoning as the send case to complete the proof that $u \cdot p \triangleleft q_1 ? m_1$ suffices as a witness for v_0 .

It is worth noting that the construction of the witness prefix v_0 in the proof immediately yields an algorithm for computing counterexample traces to implementability.

Remark 7.2 (Mixed Choice is Not Needed to Implement Global Types). Theorem 7.1 basically shows the necessity of Send Validity for implementability. Corollary 5.5 shows that Send Validity precludes states with both send and receive outgoing transitions. Together, this implies that an implementable global type can always be implemented without mixed choice. Note that the syntactic restrictions on global types do not inherently prevent mixed choice states from arising in a role’s subset construction, as evidenced by r in the following type: $p \rightarrow q : l. q \rightarrow r : m. 0 + p \rightarrow q : r. r \rightarrow q : m. 0$. Our completeness result thus implies that this type is not implementable. Most MST frameworks [18, 24, 31] implicitly force *no mixed choice* through syntactic restrictions on local types. We are the first to prove that mixed choice states are indeed not necessary for completeness. This is interesting because mixed choice is known to be crucial for the expressive power of the synchronous π -calculus compared to its asynchronous variant [32].

8 Complexity

In this section, we establish PSPACE-completeness of checking implementability for global types.

Theorem 8.1. *The MST implementability problem is PSPACE-complete.*

Proof. We first establish the upper bound. The decision procedure enumerates for each role p the subsets of $\mathbf{GAut}(\mathbf{G})\downarrow_p$. This can be done in polynomial space and exponential time. For each p and $s \subseteq Q_{\mathbf{G}}$, it then (i) checks membership of s in Q_p of $\mathcal{C}(\mathbf{G}, p)$, and (ii) if $s \in Q_p$, checks whether all outgoing transitions of s in $\mathcal{C}(\mathbf{G}, p)$ satisfy Send and Receive Validity. Check (i) can be reduced to the intersection non-emptiness problem for nondeterministic finite state machines, which is in PSPACE [44]. It is easy to see that check (ii) can be done in polynomial time. In particular, the computation of available messages for Receive Validity only requires a single unfolding of every loop in \mathbf{G} .

Note that the synthesis problem has the same complexity. The subset construction to determinize $\mathbf{GAut}(\mathbf{G})\downarrow_p$ can be done using a PSPACE transducer. While the output can be of exponential size, it is written on an extra tape that is not counted towards memory usage. However, this means we need to perform the validity checks as described above instead of using the computed deterministic state machines.

Second, we prove the lower bound. The proof is inspired by the proof for Theorem 4 [4] in which Alur et al. prove that checking safe realizability of bounded HMSCs is PSPACE-hard. We reduce the PSPACE-complete problem of checking universality of an NFA $M = (Q, \Delta, \delta, q_0, F)$ to checking implementability. Without loss of generality, we assume that every state can reach a final state. We construct a global type \mathbf{G} for p, q and r that is implementable iff $\mathcal{L}(M) = \Delta^*$. For this, we define subterms G_l and G_r as well as G_q for every $q \in Q$ and G_* . We use a fresh letter \perp to handle final states of M . We also define $p \leftrightarrow q : m$ as an abbreviation for $p \rightarrow q : m . q \rightarrow p : m$.

$$\mathbf{G} := G_l + G_r$$

$$G_l := p \leftrightarrow q : l . p \leftrightarrow r : go . G_{q_0}$$

$$G_q := \begin{cases} \sum_{(a, q') \in \delta(q)} (r \leftrightarrow q : a . G_{q'}) & \text{if } q \notin F \\ r \leftrightarrow q : \perp . 0 + \sum_{(a, q') \in \delta(q)} (r \leftrightarrow q : a . G_{q'}) & \text{if } q \in F \end{cases}$$

$$G_r := p \leftrightarrow q : r . p \leftrightarrow r : go . G_*$$

$$G_* := r \leftrightarrow q : \perp . 0 + \sum_{a \in \Delta} (r \leftrightarrow q : a . G_*)$$

The global type \mathbf{G} is constructed such that p first decides whether words from $\mathcal{L}(M)$ or from Δ^* are sent subsequently. This decision is known to p and q but not to r . The protocol then continues with r sending letters from Δ to q , and p is not involved. Intuitively, q is able to receive these letters if and only if $\mathcal{L}(M) = \Delta^*$. From Theorems 6.1 and 7.1, we know that $\{\{\mathcal{C}(\mathbf{G}, p)\}_p\}_{p \in \mathcal{P}}$ implements \mathbf{G} if \mathbf{G} is implementable.

We claim that $\{\{\mathcal{C}(\mathbf{G}, p)\}_p\}_{p \in \mathcal{P}}$ implements \mathbf{G} if and only if $\mathcal{L}(M) = \Delta^*$.

First, assume that $\mathcal{L}(M) \neq \Delta^*$. Then, there exists $w \notin \mathcal{L}(M)$. We can construct the following run of $\{\{\mathcal{C}(\mathbf{G}, \mathbf{p})\}_{\mathbf{p} \in \mathcal{P}}\}$ that deadlocks. Role \mathbf{p} chooses the left subterm G_l and, subsequently, \mathbf{r} sends w to \mathbf{q} . We do a case analysis on whether w contains a prefix w' such that $w' \notin \text{pref}(\mathcal{L}(M))$. If so, sending the last letter of a minimal prefix leads to a deadlock in $\{\{\mathcal{C}(\mathbf{G}, \mathbf{p})\}_{\mathbf{p} \in \mathcal{P}}\}$, contradicting deadlock freedom. If not, it holds that w is a prefix of a word in $\mathcal{L}(M)$. Still, role \mathbf{r} can send \perp , which cannot be received, also contradicting deadlock freedom.

Second, assume that $\mathcal{L}(M) = \Delta^*$. With this, it is fine that \mathbf{r} does not know the branch. Role \mathbf{q} will be able to receive all messages since $\mathcal{C}(\mathbf{G}, \mathbf{q})$ can receive, letter by letter, $w.\perp$ for every $w \in \mathcal{L}(M)$ from \mathbf{r} . Thus, protocol fidelity and deadlock freedom hold, concluding the proof.

Note that PSPACE-hardness only holds if the size of \mathbf{G} does not account for common subterms multiple times. Because every message is immediately acknowledged, the constructed global type specifies a universally 1-bounded [23] language, proving that PSPACE-hardness persists for such a restriction. For our construction, it does not hold that $\mathcal{V}(\mathcal{L}(G_l) \downarrow_{\Sigma_{\mathbf{q},?}}) = \mathcal{L}(M)$. We chose so to have a more compact protocol. However, we can easily fix this by sending the decision of \mathbf{r} first to \mathbf{p} , allowing to omit the messages \perp to \mathbf{q} . \square

This result and the fact that local languages are preserved by the subset projection (Lemma 4.3) leads to the following observation.

Corollary 8.2. *Let \mathbf{G} be an implementable global type. Then, the subset projection $\{\{\mathcal{P}(\mathbf{G}, \mathbf{p})\}_{\mathbf{p} \in \mathcal{P}}\}$ is a local language preserving implementation for \mathbf{G} , i.e., $\mathcal{L}(\mathcal{P}(\mathbf{G}, \mathbf{p})) = \mathcal{L}(\mathbf{G}) \downarrow_{\Sigma_{\mathbf{p}}}$ for every \mathbf{p} , and can be computed in PSPACE.*

Remark 8.3 (MST implementability with directed choice is PSPACE-hard). Theorem 8.1 is stated for global types with sender-driven choice but the provided type is in fact directed. Thus, the PSPACE lower bound also holds for implementability of types with directed choice.

9 Evaluation

We consider the following three aspects in the evaluation of our approach: (E1) difficulty of implementation (E2) completeness, and (E3) comparison to state of the art.

For this, we implemented our subset projection in a prototype tool [1, 37]. It takes a global type as input and computes the subset projection for each role. It was straightforward to implement the core functionality in approximately 700 lines of Python3 code closely following the formalization (E1).

We consider global types (and communication protocols) from seven different sources as well as all examples from this work (cf. 1st column of Table 1). Our experiments were run on a computer with an Intel Core i7-1165G7 CPU and used at most 100MB of memory. The results are summarized in Table 1. The reported size is the number of states and transitions of the respective state machine, which

Table 1. Projecting Global Types. For every protocol, we report whether it is implementable ✓ or not ✗, the time to compute our subset projection and the generalized projection by Majumdar et al. [31] as well as the outcome as ✓ for “implementable”, ✗ for “not implementable” and (✗) for “not known”. We also give the size of the protocol (number of states and transitions), the number of roles, the combined size of all subset projections (number of states and transitions).

Source	Name	Impl.	Subset Proj. (complete)	Size	$ \mathcal{P} $	Size Proj's	[31] (incomplete)
[35]	Instrument Contr. Prot. A	✓	✓ 0.4 ms	22	3	61	✓ 0.2 ms
	Instrument Contr. Prot. B	✓	✓ 0.3 ms	17	3	47	✓ 0.1 ms
	OAuth2	✓	✓ 0.1 ms	10	3	23	✓ <0.1 ms
[34]	Multi Party Game	✓	✓ 0.5 ms	21	3	67	✓ 0.1 ms
[24]	Streaming	✓	✓ 0.2 ms	13	4	28	✓ <0.1 ms
[13]	Non-Compatible Merge	✓	✓ 0.2 ms	11	3	25	✓ 0.1 ms
[45]	Spring-Hibernate	✓	✓ 1.0 ms	62	6	118	✓ 0.7 ms
[31]	Group Present	✓	✓ 0.6 ms	51	4	85	✓ 0.6 ms
	Late Learning	✓	✓ 0.3 ms	17	4	34	✓ 0.2 ms
	Load Balancer ($n = 10$)	✓	✓ 3.9 ms	36	12	106	✓ 2.4 ms
	Logging ($n = 10$)	✓	✓ 71.5 ms	81	13	322	✓ 10.0 ms
[38]	2 Buyer Protocol	✓	✓ 0.5 ms	22	3	60	✓ 0.2 ms
	2B-Prot. Omit No	✓	✓ 0.4 ms	19	3	56	(✗) 0.1 ms
	2B-Prot. Subscription	✓	✓ 0.7 ms	46	3	95	(✗) 0.3 ms
	2B-Prot. Inner Recursion	✓	✓ 0.4 ms	17	3	51	✓ 0.1 ms
New	Odd-even (Example 2.1)	✓	✓ 0.5 ms	32	3	70	(✗) 0.2 ms
	\mathbf{G}_r – Receive Val. Violated (§2)	✗	✗ 0.1 ms	12	3	-	(✗) <0.1 ms
	\mathbf{G}'_r – Receive Val. Satisfied (§2)	✓	✓ 0.2 ms	16	3	35	✓ 0.1 ms
	\mathbf{G}_s – Send Val. Violated (§2)	✗	✗ <0.1 ms	8	3	-	(✗) <0.1 ms
	\mathbf{G}'_s – Send Val. Satisfied (§2)	✓	✓ <0.1 ms	7	3	17	✓ <0.1 ms
	\mathbf{G}_{fold} (§10)	✓	✓ 0.4 ms	21	3	50	(✗) 0.1 ms
	\mathbf{G}_{unf} (§10)	✓	✓ 0.4 ms	30	3	61	✓ 0.2 ms

allows not to account for multiple occurrences of the same subterm. As expected, our tool can project every implementable protocol we have considered (E2).

Regarding the comparison against the state of the art (E3), we directly compared our subset projection to the incomplete approach by Majumdar et al. [31], and found that the run times are in the same order of magnitude in general (typically a few milliseconds). However, the projection of [31] fails to project four implementable protocols (including Example 2.1). We discuss some of the other examples in more detail in the next section. We further note that most of the run times reported by Scalas and Yoshida [36] on their model checking based tool are around 1s and are thus two to three orders of magnitude slower.

10 Discussion

Success of Syntactic Projections Depends on Representation. Let us illustrate how unfolding recursion helps syntactic projection operators to succeed. Consider this implementable global type, which is not syntactically projectable:

$$\mathbf{G}_{\text{fold}} := + \left\{ \begin{array}{l} \text{p} \rightarrow \text{q} : \text{o}. \mu t_1. (\text{p} \rightarrow \text{q} : \text{o}. \text{q} \rightarrow \text{r} : \text{o}. t_1 + \text{p} \rightarrow \text{q} : \text{b}. \text{q} \rightarrow \text{r} : \text{b}. 0) \\ \text{p} \rightarrow \text{q} : \text{m}. \text{q} \rightarrow \text{r} : \text{m}. \mu t_2. (\text{p} \rightarrow \text{q} : \text{o}. \text{q} \rightarrow \text{r} : \text{o}. t_2 + \text{p} \rightarrow \text{q} : \text{b}. \text{q} \rightarrow \text{r} : \text{b}. 0) \end{array} \right. .$$

Similar to projection by erasure, a syntactic projection erases events that a role is not involved in and immediately tries to *merge* different branches. The merge operator is a partial operator that checks sufficient conditions for implementability. Here, the merge operator fails for r because it cannot merge a recursion variable binder and a message reception. Unfolding the global type preserves the represented protocol and resolves this issue:

$$\mathbf{G}_{\text{unf}} := + \left\{ \begin{array}{l} \text{p} \rightarrow \text{q} : \text{o}. \left\{ \begin{array}{l} \text{p} \rightarrow \text{q} : \text{b}. \text{q} \rightarrow \text{r} : \text{b}. 0 \\ \text{p} \rightarrow \text{q} : \text{o}. \text{q} \rightarrow \text{r} : \text{o}. \mu t_1. (\text{p} \rightarrow \text{q} : \text{o}. \text{q} \rightarrow \text{r} : \text{o}. t_1 + \text{p} \rightarrow \text{q} : \text{b}. \text{q} \rightarrow \text{r} : \text{b}. 0) \end{array} \right. \\ \text{p} \rightarrow \text{q} : \text{m}. \text{q} \rightarrow \text{r} : \text{m}. \mu t_2. (\text{p} \rightarrow \text{q} : \text{o}. \text{q} \rightarrow \text{r} : \text{o}. t_2 + \text{p} \rightarrow \text{q} : \text{b}. \text{q} \rightarrow \text{r} : \text{b}. 0) \end{array} \right. .$$

(We refer to [29] for visual representations of both global types.) This global type can be projected with most syntactic projection operators and shows that the representation of the global type matters for syntactic projectability. However, such unfolding tricks do not always work, e.g. for the odd-even protocol (Example 2.1). We avoid this brittleness using automata and separating the synthesis from checking implementability.

Entailed Properties from the Literature. We defined implementability for a global type as the question of whether there exists a deadlock-free CSM that generates the same language as the global type. Various other properties of implementations and protocols have been proposed in the literature. Here, we give a brief overview and defer to the extended version [29] for a detailed analysis. *Progress* [18], a common property, requires that every sent message is eventually received and every expected message will eventually be sent. With deadlock freedom, our subset projection trivially satisfies progress for finite traces. For infinite traces, as expected, fairness assumptions are required to enforce progress. Similarly, our subset projection prevents *unspecified receptions* [14] and *orphan messages* [9, 21], respectively interpreted in our multiparty setting with sender-driven choice. We also ensure that every local transition of each role is *executable* [14], i.e. it is taken in some run of the CSM. Any implementation of a global type has the *stable property* [28], i.e., one can always reach a configuration with empty channels from every reachable configuration. While the properties above are naturally satisfied by our subset projection, the following ones can be checked directly on an implementable global type without explicitly constructing the implementation. A global type is *terminating* [36] iff it does not contain recursion and *never-terminating* [36] iff it does not contain term 0.

11 Related Work

MSTs were introduced by Honda et al. [24] with a process algebra semantics, and the connection to CSMs was established soon afterwards [20].

In this work, we present a complete projection procedure for global types with sender-driven choice. The work by Castagna et al. [13] is the only one to present a projection that aims for completeness. Their semantic conditions, however, are not effectively computable and their notion of completeness is “less demanding than the classical ones” [13]. They consider multiple implementations, generating different sets of traces, to be sound and complete with regard to a single global type [13, Sec. 5.3]. In addition, the algorithmic version of their conditions does not use global information as our message availability analysis does.

MST implementability relates to safe realizability of HMSCs, which is undecidable in general but decidable for certain classes [30]. Stutz [38] showed that implementability of global types that are always able to terminate is decidable.¹ The EXPSPACE decision procedure is obtained via a reduction to safe realizability of globally-cooperative HMSCs, by proving that the HMSC encoding [39] of any implementable global type is globally-cooperative and generalizing results for infinite executions. Thus, our PSPACE-completeness result both generalizes and tightens the earlier decidability result obtained in [38]. Stutz [38] also investigates how HMSC techniques for safe realizability can be applied to the MST setting – using the formal connection between MST implementability and safe realizability of HMSCs – and establishes an undecidability result for a variant of MST implementability with a relaxed indistinguishability relation.

Similar to the MST setting, there have been approaches in the HMSC literature that tie branching to a role making a choice. We refer the reader to the work by Majumdar et al. [31] for a survey.

Standard MST frameworks project a global type to a set of *local types* rather than a CSM. Local types are easily translated to FSMs [31, Def.11]. Our projection operator, though, can yield FSMs that cannot be expressed with the limited syntax of local types. Consider this implementable global type: $p \rightarrow q : o. 0 + p \rightarrow q : m. p \rightarrow r : b. 0$. The subset projection for r has two final states connected by a transition labeled $r \langle p ? b$. In the syntax of local types, 0 is the only term indicating termination, which means that final states with outgoing transitions cannot be expressed. In contrast to the syntactic restrictions for global types, which are key to effective verification, we consider local types unnecessarily restrictive. Usually, local implementations are type-checked against their local types and subtyping gives some implementation freedom [12, 16, 17, 27]. However, one can also view our subset projection as a local specification of the actual implementation. We conjecture that subtyping would then amount to a variation of alternating refinement [5].

CSMs are Turing-powerful [11] but decidable classes were obtained for different semantics: restricted communication topology [33, 42], half-duplex communication (only for two roles) [14], input-bounded [10], and unreliable channels [2, 3].

¹ This syntactic restriction is referred to as 0-reachability in [38].

Global types (as well choreography automata [7]) can only express existentially 1-bounded, 1-synchronizable and half-duplex communication [39]. Key to this result is that sending and receiving a message is specified atomically in a global type — a feature Dagnino et al. [19] waived for their deconfined global types. However, Dagnino et al. [19] use deconfined types to capture the behavior of a given system rather than projecting to obtain a system that generates specified behaviors.

This work relies on reliable communication as is standard for MST frameworks. Work on fault-tolerant MST frameworks [8, 43] attempts to relax this restriction. In the setting of reliable communication, both context-free [25, 40] and parametric [15, 22] versions of session types have been proposed to capture more expressive protocols and entire protocol families respectively. Extending our approach to these generalizations is an interesting direction for future work.

Acknowledgements. This work is funded in part by the National Science Foundation under grant 1815633. Felix Stutz was supported by the Deutsche Forschungsgemeinschaft project 389792660 TRR 248—CPEC.

References

1. Prototype Implementation of Subset Projection for Multiparty Session Types. <https://gitlab.mpi-sws.org/fstutz/async-mpst-gen-choice/>
2. Abdulla, P.A., Aiswarya, C., Atig, M.F.: Data communicating processes with unreliable channels. In: Grohe, M., Koskinen, E., Shankar, N. (eds.) Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, 5–8 July 2016, pp. 166–175. ACM (2016). <https://doi.org/10.1145/2933575.2934535>
3. Abdulla, P.A., Bouajjani, A., Jonsson, B.: On-the-fly analysis of systems with unbounded, lossy FIFO channels. In: Hu, A.J., Vardi, M.Y. (eds.) CAV 1998. LNCS, vol. 1427, pp. 305–318. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0028754>
4. Alur, R., Etessami, K., Yannakakis, M.: Realizability and verification of MSC graphs. *Theor. Comput. Sci.* **331**(1), 97–114 (2005). <https://doi.org/10.1016/j.tcs.2004.09.034>
5. Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.Y.: Alternating refinement relations. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 163–178. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0055622>
6. Ancona, D., et al.: Behavioral types in programming languages. *Found. Trends Program. Lang.* **3**(2-3), 95–230 (2016). <https://doi.org/10.1561/25000000031>
7. Barbanera, F., Lanese, I., Tuosto, E.: Choreography automata. In: Bliudze, S., Bocchi, L. (eds.) COORDINATION 2020. LNCS, vol. 12134, pp. 86–106. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50029-0_6
8. Barwell, A.D., Scalas, A., Yoshida, N., Zhou, F.: Generalised multiparty session types with crash-stop failures. In: Klin, B., Lasota, S., Muscholl, A. (eds.) 33rd International Conference on Concurrency Theory, CONCUR 2022, 12–16 September 2022, Warsaw, Poland. LIPIcs, vol. 243, pp. 35:1–35:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). <https://doi.org/10.4230/LIPIcs.CONCUR.2022.35>

9. Bocchi, L., Lange, J., Yoshida, N.: Meeting deadlines together. In: Aceto, L., de Frutos-Escrig, D. (eds.) 26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, 1–4 September 2015. LIPIcs, vol. 42, pp. 283–296. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015). <https://doi.org/10.4230/LIPIcs.CONCUR.2015.283>
10. Bollig, B., Finkel, A., Suresh, A.: Bounded reachability problems are decidable in FIFO machines. In: Konnov, I., Kovács, L. (eds.) 31st International Conference on Concurrency Theory, CONCUR 2020, 1–4 September 2020, Vienna, Austria (Virtual Conference). LIPIcs, vol. 171, pp. 49:1–49:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). <https://doi.org/10.4230/LIPIcs.CONCUR.2020.49>
11. Brand, D., Zafropulo, P.: On communicating finite-state machines. *J. ACM* **30**(2), 323–342 (1983). <https://doi.org/10.1145/322374.322380>
12. Bravetti, M., Carbone, M., Zavattaro, G.: On the boundary between decidability and undecidability of asynchronous session subtyping. *Theor. Comput. Sci.* **722**, 19–51 (2018). <https://doi.org/10.1016/j.tcs.2018.02.010>
13. Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On global types and multiparty session. *Log. Methods Comput. Sci.* **8**(1) (2012). [https://doi.org/10.2168/LMCS-8\(1:24\)2012](https://doi.org/10.2168/LMCS-8(1:24)2012)
14. Cécé, G., Finkel, A.: Verification of programs with half-duplex communication. *Inf. Comput.* **202**(2), 166–190 (2005). <https://doi.org/10.1016/j.ic.2005.05.006>
15. Charalambides, M., Dinges, P., Agha, G.A.: Parameterized, concurrent session types for asynchronous multi-actor interactions. *Sci. Comput. Program.* **115–116**, 100–126 (2016). <https://doi.org/10.1016/j.scico.2015.10.006>
16. Chen, T., Dezani-Ciancaglini, M., Scalas, A., Yoshida, N.: On the preciseness of subtyping in session types. *Log. Methods Comput. Sci.* **13**(2) (2017). [https://doi.org/10.23638/LMCS-13\(2:12\)2017](https://doi.org/10.23638/LMCS-13(2:12)2017)
17. Chen, T., Dezani-Ciancaglini, M., Yoshida, N.: On the preciseness of subtyping in session types. In: Chitil, O., King, A., Danvy, O. (eds.) Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, 8–10, September 2014. pp. 135–146. ACM (2014). <https://doi.org/10.1145/2643135.2643138>
18. Coppo, M., Dezani-Ciancaglini, M., Padovani, L., Yoshida, N.: A gentle introduction to multiparty asynchronous session types. In: Bernardo, M., Johnsen, E.B. (eds.) SFM 2015. LNCS, vol. 9104, pp. 146–178. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18941-3_4
19. Dagnino, F., Giannini, P., Dezani-Ciancaglini, M.: Deconfined global types for asynchronous sessions. In: Damiani, F., Dardha, O. (eds.) COORDINATION 2021. LNCS, vol. 12717, pp. 41–60. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78142-2_3
20. Deniérou, P.-M., Yoshida, N.: Multiparty session types meet communicating automata. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 194–213. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_10
21. Deniérou, P.-M., Yoshida, N.: Multiparty compatibility in communicating automata: characterisation and synthesis of global session types. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013. LNCS, vol. 7966, pp. 174–186. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39212-2_18
22. Deniérou, P., Yoshida, N., Bejleri, A., Hu, R.: Parameterised multiparty session types. *Log. Methods Comput. Sci.* **8**(4) (2012). [https://doi.org/10.2168/LMCS-8\(4:6\)2012](https://doi.org/10.2168/LMCS-8(4:6)2012)

23. Genest, B., Kuske, D., Muscholl, A.: On communicating automata with bounded channels. *Fundam. Inform.* **80**(1–3), 147–167 (2007). <http://content.iospress.com/articles/fundamenta-informaticae/fi80-1-3-09>
24. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Necula, G.C., Wadler, P. (eds.) *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*, San Francisco, California, USA, 7–12 January 2008, pp. 273–284. ACM (2008). <https://doi.org/10.1145/1328438.1328472>
25. Keizer, A.C., Basold, H., Pérez, J.A.: Session coalgebras: a coalgebraic view on regular and context-free session types. *ACM Trans. Program. Lang. Syst.* **44**(3), 18:1–18:45 (2022). <https://doi.org/10.1145/3527633>
26. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **21**(7), 558–565 (1978). <https://doi.org/10.1145/359545.359563>
27. Lange, J., Yoshida, N.: On the undecidability of asynchronous session subtyping. In: Esparza, J., Murawski, A.S. (eds.) *FoSSaCS 2017*. LNCS, vol. 10203, pp. 441–457. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54458-7_26
28. Lange, J., Yoshida, N.: Verifying asynchronous interactions via communicating session automata. In: Dillig, I., Tasiran, S. (eds.) *CAV 2019*. LNCS, vol. 11561, pp. 97–117. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_6
29. Li, E., Stutz, F., Wies, T., Zufferey, D.: Complete multiparty session type projection with automata. *CoRR* abs/2305.17079 (2023). <https://doi.org/10.48550/arXiv.2305.17079>
30. Lohrey, M.: Realizability of high-level message sequence charts: closing the gaps. *Theor. Comput. Sci.* **309**(1–3), 529–554 (2003). <https://doi.org/10.1016/j.tcs.2003.08.002>
31. Majumdar, R., Mukund, M., Stutz, F., Zufferey, D.: Generalising projection in asynchronous multiparty session types. In: Haddad, S., Varacca, D. (eds.) *32nd International Conference on Concurrency Theory, CONCUR 2021*, 24–27 August 2021, Virtual Conference. LIPIcs, vol. 203, pp. 35:1–35:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). <https://doi.org/10.4230/LIPIcs.CONCUR.2021.35>
32. Palamidessi, C.: Comparing the expressive power of the synchronous and asynchronous pi-calculi. *Math. Struct. Comput. Sci.* **13**(5), 685–719 (2003). <https://doi.org/10.1017/S09601295030004043>
33. Peng, W., Purushothaman, S.: Analysis of a class of communicating finite state machines. *Acta Informatica* **29**(6/7), 499–522 (1992). <https://doi.org/10.1007/BF01185558>
34. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: Müller, P. (ed.) *31st European Conference on Object-Oriented Programming, ECOOP 2017*, 19–23 June 2017, Barcelona, Spain. LIPIcs, vol. 74, pp. 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). <https://doi.org/10.4230/LIPIcs.ECOOP.2017.24>
35. Scalas, A., Yoshida, N.: *Mpstk: the multiparty session types toolkit* (2018). <https://doi.org/10.1145/3291638>
36. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.* **3**(POPL), 30:1–30:29 (2019). <https://doi.org/10.1145/3290343>
37. Stutz, F.: Artifact for “Complete Multiparty Session Type Projection with Automata”, April 2023. <https://doi.org/10.5281/zenodo.7878493>

38. Stutz, F.: Asynchronous multiparty session type implementability is decidable - lessons learned from message sequence charts. In: 37th European Conference on Object-Oriented Programming, ECOOP 2023. LIPIcs (2023). <https://arxiv.org/pdf/2302.11272.pdf>
39. Stutz, F., Zufferey, D.: Comparing channel restrictions of communicating state machines, high-level message sequence charts, and multiparty session types. In: Ganty, P., Monica, D.D. (eds.) Proceedings of the 13th International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2022, Madrid, Spain, 21–23 September 2022. EPTCS, vol. 370, pp. 194–212 (2022). <https://doi.org/10.4204/EPTCS.370.13>
40. Thiemann, P., Vasconcelos, V.T.: Context-free session types. In: Garrigue, J., Keller, G., Sumii, E. (eds.) Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, 18–22 September 2016, pp. 462–475. ACM (2016). <https://doi.org/10.1145/2951913.2951926>
41. Toninho, B., Yoshida, N.: Certifying data in multiparty session types. *J. Log. Algebraic Methods Program.* **90**, 61–83 (2017). <https://doi.org/10.1016/j.jlamp.2016.11.005>
42. La Torre, S., Madhusudan, P., Parlato, G.: Context-bounded analysis of concurrent queue systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 299–314. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_21
43. Viering, M., Hu, R., Eugster, P., Ziarek, L.: A multiparty session typing discipline for fault-tolerant event-driven distributed programming. *Proc. ACM Program. Lang.* **5**(OOPSLA), 1–30 (2021). <https://doi.org/10.1145/3485501>
44. Wehar, M.: On the complexity of intersection non-emptiness problems. Ph.D. thesis, University of Buffalo (2016)
45. Spring and Hibernate Transaction in Java. <https://www.uml-diagrams.org/examples/spring-hibernate-transaction-sequence-diagram-example.html>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

