Characterizing Implementability of Global Protocols with Infinite States and Data

ELAINE LI*, New York University, USA
FELIX STUTZ, University of Luxembourg, Luxembourg
THOMAS WIES, New York University, USA
DAMIEN ZUFFEREY†, NVIDIA, Switzerland

We study the implementability problem for an expressive class of symbolic communication protocols involving multiple participants. Our symbolic protocols describe infinite states and data values using dependent refinement predicates. Implementability asks whether a global protocol specification admits a distributed, asynchronous implementation, namely one for each participant, that is deadlock-free and exhibits the same behavior as the specification. We provide a unified explanation of seemingly disparate sources of non-implementability through a precise semantic characterization of implementability for infinite protocols. Our characterization reduces the problem of implementability to (co)reachability in the global protocol restricted to each participant. This compositional reduction yields the first sound and relatively complete algorithm for checking implementability of symbolic protocols. We use our characterization to show that for finite protocols, implementability is co-NP-complete for explicit representations and PSPACE-complete for symbolic representations. The finite, explicit fragment subsumes a previously studied fragment of multiparty session types for which our characterization yields a co-NP decision procedure, tightening a prior PSPACE upper bound.

Additional Key Words and Phrases: Protocol verification, Multiparty session types, Refinement

1 Introduction

Concurrency is ubiquitous in modern computing, message-passing is a major concurrency paradigm, and communication protocols are therefore a key target for formal verification. Communication protocols specify distributed, message-passing behaviors from a global point of view, altogether describing the interactions between all participants in the protocol. Implementability and synthesis are two central questions to the verification of communication protocols. Implementability asks whether a protocol admits a distributed implementation, and synthesis in turn computes an admissible one. A distributed implementation is considered admissible if it is deadlock-free and exhibits exactly the same communication behaviors described by the specification. We refer to the latter property as *protocol fidelity*. The implementability question precedes the synthesis question in importance: synthesizing implementations for unrealizable protocols is a fruitless endeavor.

Global protocol specifications find industry applications in the form of UML's high-level message sequence charts and the Web Service Choreography Description Language, and are widely studied in academia in the form of multiparty session types and choreographic programming. Multiparty

*corresponding author
†Damien Zufferey was working at SonarSource in Switzerland when this work began.

Authors' Contact Information: Elaine Li, New York University, New York, USA, efl9013@nyu.edu; Felix Stutz, University of Luxembourg, Esch-sur-Alzette, Luxembourg, felix.stutz@uni.lu; Thomas Wies, New York University, New York, USA, wies@cs.nyu.edu; Damien Zufferey, NVIDIA, Zurich, Switzerland, rilaak@gmail.com.

ACM 2475-1421/2025/4-ART131
https://doi.org/10.1145/3720493

session types (MSTs) have been implemented in at least 16 programming languages including Python [25, 69, 71], Java [46, 47], C [72], Go [11, 54], Scala [12], Rust [16, 52], OCaml [48], F# [70], and applied to operating systems [28], high performance computing [24, 44, 73], cyber-physical systems [62, 63], and web services [87]. We refer the reader to [86] and [65] for a comprehensive survey of MST and choreography applicability respectively.

To model real-world verification targets, we desire for our protocol specifications to be as expressive as possible. Various dimensions of expressivity have been explored in the literature, such as arbitrary message payloads, non-deterministic choice, unrestricted recursion and parametricity. Formalisms such as choreography automata [34], high-level message sequence charts [1, 3, 30–33, 59, 64, 66, 68, 74] and session types [6, 7, 45, 56, 81, 89] correspond to syntactically-defined fragments that incorporate a selection of these features.

In this paper, we study the implementability problem for a semantically-defined class of communication protocols, which we call *global communicating labeled transition systems (GCLTS)*. GCLTS impose only modest syntactic restrictions and subsume many existing fragments of asynchronous multiparty session types and choreography automata. GCLTS capture the following important features:

- Asynchrony: the semantics are interpreted over a peer-to-peer, asynchronous network, with FIFO channels connecting each pair of protocol participants.
- Generalized sender-driven choice: the only notable syntactic restriction imposed by our formalism is that at each branching point of the protocol's control flow, a single participant chooses a branch. In other words, the first message that is sent in each branch of a choice must come from the same sender. However, we impose no restrictions on the recipient or the message payload other than that no two branches share the same recipient and message.
- Infinite protocol state: protocol states contain registers that take values from an infinite domain. This allows loops to carry memory across iterations, and allows the protocol to be specified in terms of dependent refinement predicates.
- Infinite message payloads: messages can carry values drawn from an infinite data domain.

Implementability is undecidable for this general class of protocols. The presence of and interaction between the aforementioned features means that even soundly approximating implementability is challenging. Existing work is either comparable in expressivity but does not solve the implementability problem, or solves the implementability problem but is incomparably restricted in expressivity. Zhou et al. [89] present a framework for synchronous, refined multiparty session types that soundly approximates implementability through its endpoint projection, but that may yield local specifications that are not implementable. Several works [2, 56, 59, 78] precisely characterize implementability for finite protocol specifications. However, the implementability check in [2, 56] relies on synthesizing an implementation upfront, which is not possible for infinite-state protocols. Das and Pfenning [22] study local session types with arithmetic refinements in a binary setting.

We address these challenges by decomposing the implementability problem into two steps. First, we give a precise, semantic characterization of implementability for GCLTS that we prove sound and complete once and for all. Our characterization is defined directly on the global specification, and thus forgoes the need to first synthesize a candidate implementation. Moreover, our characterization gives a unified semantic explanation to disparate causes of non-implementability that arise from the expressivity of our protocol fragment. We encapsulate the complexities introduced by communication-specific features such as asynchrony and partial information in the first step. Our semantic characterization reduces implementability to (co)reachability in the GCLTS. Specifically, we provide a sound and complete reduction to the first-order fixpoint logic μCLP [83]. The μCLP calculus can express recursive predicates with least and greatest fixpoint semantics where

the predicate body is constrained by a first-order logic formula over a background theory. Our implementability characterization can therefore be checked by existing μCLP solvers. Second, we use this reduction to obtain a blueprint for solving implementability algorithmically. Our reduction yields algorithms that are sound and complete relative to an assumed oracle for solving μCLP validity, in addition to decision procedures with optimal complexity for various decidable classes.

Contributions. In summary, our contributions are:

- Global communicating labeled transition systems (GCLTS): a semantically-defined class of asynchronous communication protocols that subsumes most formalisms in the literature.
- A precise characterization of implementability for GCLTS.
- The first symbolic algorithm for checking implementability of infinite, symbolic protocols that is sound and relatively complete.
- Optimal decision procedures for checking implementability of finite protocols. In particular, we show that for explicit protocol representations that enumerate all states and transitions, the problem is co-NP-complete, and for symbolic protocol representations that encode states and transitions using predicates and variables, the problem is PSPACE-complete.
- As a corollary of the previous result, we obtain a co-NP decision procedure for implementability of global types, tightening a prior PSPACE upper bound [56, 57].

2 Overview

We motivate our work using an infinite state version of a two-bidder protocol, depicted as a high-level message sequence chart (HMSC) in Fig. 1. The protocol specifies the behavior of two bidders, B_1 and B_2, who negotiate to split the purchase of a book from seller S.

The protocol begins with B_1 announcing to S and B_2 the book y it proposes to buy. The protocol requires that y signifies a valid ISBN number, which we abstract with the predicate $\mathsf{ISBN}(y)$. The seller S then informs B_1 the requested book's price z. After this, B_1 and B_2 enter a bidding phase in which they negotiate the split of their respective contributions b_1 and b_2 towards the purchase. In each round of the bidding phase, B_1 proposes its contribution b_1 to B_2. Bidder B_2 then decides to either abort the protocol by sending a quit message to S, or respond to B_1 with its own bid b_2. In case B_2 aborts, S echoes the abort message to B_1 and the protocol terminates. In case B_2 continues bidding, if the sum of the proposed bids exceeds the book's price, B_1 informs S of the successful negotiation. Seller S in turn relays the message to B_2. Otherwise, B_1 sends a cont message to B_2, informing them that they need to enter another bidding round. Throughout the bidding phase, B_1 and B_2 track the values of their latest bids in the registers z_1 and z_2. The refinements ensure that the proposed bids are strictly increasing from one round to the next, thus enforcing termination.

Figs. 2 to 4 show an admissible implementation for the two-bidder protocol in Fig. 1, consisting of a local implementation for each participant: S, B_1 and B_2. The transition labels specify their local behaviors: $\mathsf{B}_1 \triangleright \mathsf{S}!y\{\mathsf{ISBN}(y)\}$ specifies that B_1 sends a message y to S such that y satisfies $\mathsf{ISBN}(y)$, i.e. y is a valid ISBN number; $\mathsf{S} \triangleleft \mathsf{B}_1?y\{\mathsf{ISBN}(y)\}$ specifies that S receives y from B_1, and can assume $\mathsf{ISBN}(y)$ holds of y. We assume an asynchronous setting in which every pair of participants is connected by a FIFO channel. The implementability of Fig. 1 is witnessed by Figs. 2 to 4, which together exhibit the same behaviors as the global protocol and is never stuck.

To see that the implementability problem is non-trivial, consider a variant of the protocol in Fig. 1 where the succ message to S is sent by B_2 instead of B_1. The resulting protocol is no longer implementable because B_2 never learns about the price z of the book y and is therefore unable to determine when the negotiation with B_1 has succeeded.

Our example highlights several important expressive features of GCLTS:

- Generalized sender-driven choice: after B_2 receives a bid from B_1, it has the option to either send a bid back to B_1 and continue the bidding process, or terminate the protocol by sending a quit message to the bookseller, who then relays the termination message to the first bidder. Due to this choice interaction alone, the protocol is not expressible in [89].
- Infinite state: the protocol state contains registers that can be assigned values from an infinite domain. Registers are updated to store the last bid from each round z_1 and z_2, and to enforce that bidders make strictly increasing bids per round.
- Infinite message data: message payload values can be drawn from an infinite data domain, such as the book price z and bids b_1 and b_2.
- Dependent refinement predicates: message payloads are constrained by data refinements such as $z_1 < b_1$ and $z < b_1 + b_2$. The refinement predicates can refer to current register values in addition to data values sent in prior messages.
- Partial information: each protocol participant only has a partial view of the global protocol state. For example, even though S participates in the bidding phase of the protocol, it never learns about the bids b_1 and b_2 in each bidding round. In fact, the registers z_1 and z_2 that store the last bid are known only to the bidders.

The presence of these features in the class of communication protocols we consider makes checking implementability uniquely challenging. For protocols with finite GCLTS specifications, deciding implementability in the presence of asynchrony and non-deterministic choice already presents a challenge. Note that finiteness here refers only to the specification, and does not mean that the underlying protocol is finite-state, nor that it contains only finite traces. Most existing work has therefore focused on developing projection operators that are sound but incomplete [14, 45, 75, 81]. These projection operators solve implementability and synthesis simultaneously by computing a candidate implementation, but often fail eagerly for protocols for which an implementation exists. Li et al. [56] proposed the first sound and complete projection operator for finite, asynchronous, multiparty session types. The projection operator critically relies on the observation that if a global type is implementable, then a canonical implementation implements it. Thus, the implementability

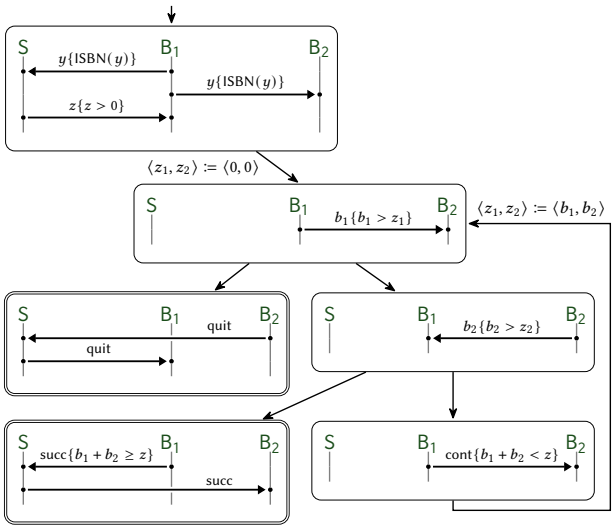

Fig. 1. Two-bidder protocol.

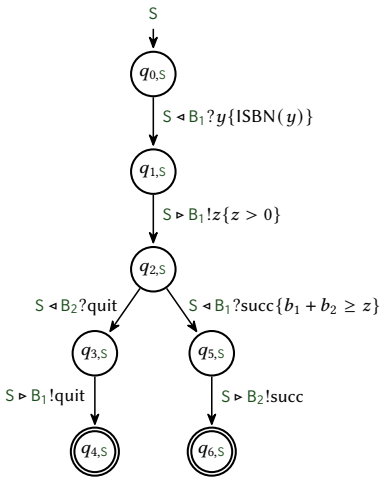

Fig. 2. State machine for seller S for Fig. 1.

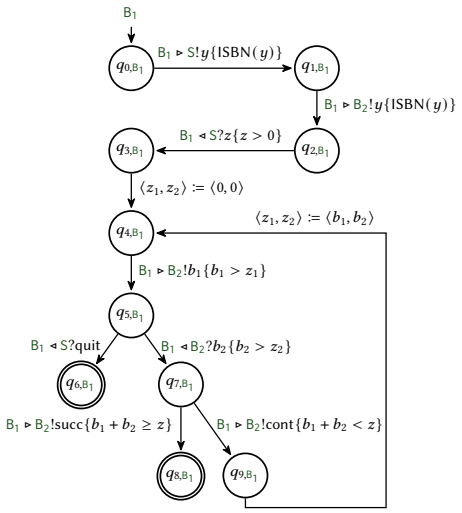

Fig. 3. State machine for bidder B_1 for Fig. 1.

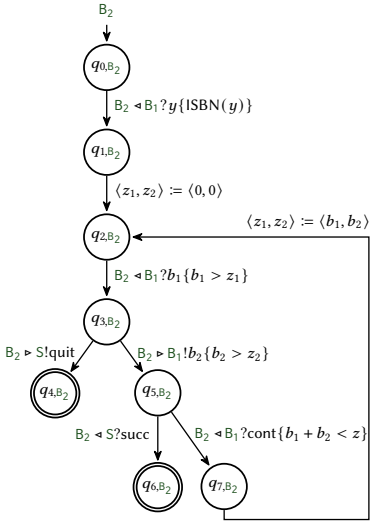

Fig. 4. State machine for bidder B_2 for Fig. 1.

problem reduces to checking whether this canonical implementation indeed implements the global type, i.e. it recognizes the same set of behaviors and is deadlock-free. Towards these ends, Li et al. [56] identify sound and complete conditions, referred to as *Send Validity* and *Receive Validity*, that are checked on the states of the canonical implementation.

In the presence of dependent refinement predicates, checking these conditions is not straightforward. Consider the examples $\mathbb{S}_1$ (using ⓐ) and $\mathbb{S}_1'$ (using ⓑ) in Fig. 5, which are variations of the examples for Receive Validity [56] featuring dependent predicates. A transition label $\mathsf{p} \to \mathsf{r} : y\{y > x\}$, which is ⓐ for $\mathbb{S}_1$, atomically specifies the send event by p and the corresponding receive event by r, along with the constraint that y satisfies $y > x$. In $\mathbb{S}_1$, participant p chooses a branch without explicitly informing r of their choice. In both branches, r is required to subtract the second value that is sent from the first value that is sent, and send the result back to p. However, due to asynchrony, both messages can arrive in r's message channels simultaneously, and r cannot tell which value was sent first. Therefore, r may subtract the values in the wrong order, rendering the protocol unimplementable.

Li et al. [56] propose one method of protocol repair: introducing a message sent by r on each branch that creates a causal dependency between the messages from p and q, so that r can no longer receive them in either order. The incorporation of dependent refinements enables a new method of protocol repair: one that does not change the communication events among the participants. The newly repaired protocol is depicted in $\mathbb{S}_1'$, in which the predicate on the second transition is changed from $y > x$ to $y = x + 2$. Despite the fact that r is still not informed of p's choice, r can

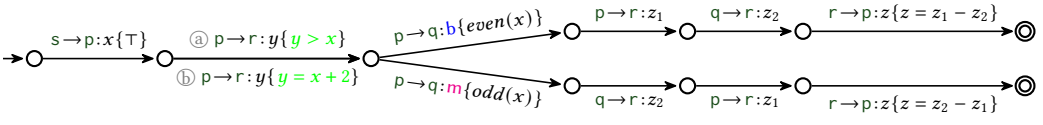

Fig. 5. Two protocols: $\mathbb{S}_1$ using ⓐ with receive order violation $\mathbb{S}_1'$ using ⓑ without receive order violation.

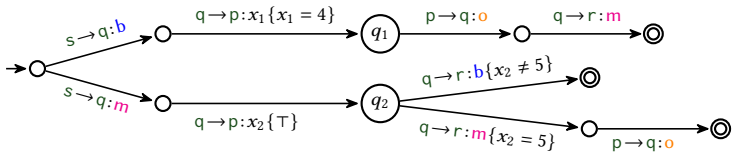

Fig. 6. $\mathbb{S}_2$: An protocol with a send violation.

infer p's choice through the parity of the first value it received from p and thus correctly follow the protocol: if y is even, r receives from p first, and if y is odd, r receives from q first.

We now turn our attention to send violations. In the protocol shown in Fig. 6, s chooses a branch and communicates its choice to q. Participant p is again not explicitly informed of the choice: in fact, p can receive 4 from q on both branches. At first glance, it appears as though it is safe for p to send o to q upon receiving 4 from q, because whilst p cannot distinguish the two branches, both branches contain the transition $\mathsf{p} \to \mathsf{q} : \mathsf{o}$. Upon closer inspection, the predicate guarding the transition immediately preceding $\mathsf{p} \to \mathsf{q} : \mathsf{o}$ on the lower branch, $x_2 = 5$, is only satisfied when p receives 5 from q. When p receives 4, the lower branch from q_2 is disabled, and since the upper branch from q_2 does not contain the transition $\mathsf{p} \to \mathsf{q} : \mathsf{o}$, the protocol is not implementable.

The examples above exemplify the ways in which refinement predicates complicate implementability checking for symbolic protocols. We return to these examples, in addition to some others, in greater detail in §4 when we present our precise characterization of implementability. We structure the rest of the paper as follows. §3 presents relevant preliminary definitions and defines the class of communication protocols we consider. §4 presents our semantic characterization of implementability for GCLTS in terms of (co)reachability, and proves that it is precise. §5 describes our sound and complete reduction from the characterization in §4 to logical formulas in μCLP [83], and additionally presents improved complexity results under certain finiteness assumptions on the GCLTS. §6 discusses related work and concludes.

3 Preliminaries

We introduce some basic concepts and notation before defining our class of protocols.

Words. Let Σ be an 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 $u \preceq v$, if there exists $w \in \Sigma^\infty$ with $u \cdot w = v$; we denote all prefixes of u with $\mathrm{pref}(u)$. Given a word $w = w_0 \dots w_n$, we use $w[i]$ to denote the i-th symbol $w_i \in \Sigma$, and $w[0..i]$ to denote the subword between and including w_0 and w_i, i.e. $w_0 \dots w_i$.

Message Alphabets. Let $\mathcal{P}$ be a finite set of participants and $\mathcal{V}$ be a (possibly infinite) data domain. We define the set of *synchronous events* $\Gamma_{sync} \coloneqq \{\mathsf{p} \to \mathsf{q} : m \mid \mathsf{p}, \mathsf{q} \in \mathcal{P} \text{ and } m \in \mathcal{V}\}$ where $\mathsf{p} \to \mathsf{q} : m$ denotes a message exchange of m from sender p to receiver q. For a participant $\mathsf{p} \in \mathcal{P}$, we define the alphabet $\Gamma_\mathsf{p} = \{\mathsf{p} \to \mathsf{q} : m \mid \mathsf{q} \in \mathcal{P},\ m \in \mathcal{V}\} \cup \{\mathsf{q} \to \mathsf{p} : m \mid \mathsf{q} \in \mathcal{P},\ m \in \mathcal{V}\}$, and a homomorphism $\Downarrow_{\Gamma_\mathsf{p}}$, where $x\Downarrow_{\Gamma_\mathsf{p}} = x$ if $x \in \Gamma_\mathsf{p}$ and ε otherwise. A synchronous event is split into a send and receive event for the respective participant, yielding *asynchronous events*. For a participant $\mathsf{p} \in \mathcal{P}$, we define the alphabet $\Sigma_{\mathsf{p},!} = \{\mathsf{p} \triangleright \mathsf{q}!m \mid \mathsf{q} \in \mathcal{P},\ m \in \mathcal{V}\}$ of *send* events and the alphabet $\Sigma_{\mathsf{p},?} = \{\mathsf{p} \triangleleft \mathsf{q}?m \mid \mathsf{q} \in \mathcal{P},\ m \in \mathcal{V}\}$ of *receive* events. The event $\mathsf{p} \triangleright \mathsf{q}!m$ denotes participant p sending a message m to q, and $\mathsf{p} \triangleleft \mathsf{q}?m$ denotes participant p receiving a message m from q. We write $\Sigma_\mathsf{p} = \Sigma_{\mathsf{p},!} \cup \Sigma_{\mathsf{p},?}$, $\Sigma_! = \bigcup_{\mathsf{p}\in\mathcal{P}} \Sigma_{\mathsf{p},!}$, and $\Sigma_? = \bigcup_{\mathsf{p}\in\mathcal{P}} \Sigma_{\mathsf{p},?}$. Finally, $\Sigma_{async} = \Sigma_! \cup \Sigma_?$. We define a homomorphism to map the synchronous alphabet to its asynchronous counterpart, defined as $\mathtt{split}(\mathsf{p} \to \mathsf{q} : m) \coloneqq \mathsf{p} \triangleright \mathsf{q}!m.\ \mathsf{q} \triangleleft \mathsf{p}?m$. Because split is injective, there exists a unique inverse,

which we denote $\texttt{split}^{-1}$. We say that p is *active* in $x \in \Sigma_{async}$ if $x \in \Sigma_{\mathsf{p}}$. For each participant $\mathsf{p} \in \mathcal{P}$, we define a homomorphism $\Downarrow_{\Sigma_{\mathsf{p}}}$, where $x\Downarrow_{\Sigma_{\mathsf{p}}} = x$ if $x \in \Sigma_{\mathsf{p}}$ 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 remainder of the paper.

Labeled Transition Systems. A *labeled transition system* (LTS) is a tuple $\mathcal{S} = (S, \Gamma, T, s_0, F)$ where S is a set of states, Γ is a set of labels, T is a set of transitions from $S \times \Gamma \times S$, $F \subseteq S$ is a set of final states, and $s_0 \in S$ is the initial state. We use $p \xrightarrow{\alpha} q$ to denote the transition $(p, \alpha, q) \in T$. Runs and traces of an LTS are defined in the expected way. A run is *maximal* if it is either finite and ends in a final state, or is infinite. The language of an LTS $\mathcal{S}$, denoted $\mathcal{L}(\mathcal{S})$, is defined as the set of maximal traces. A state $s \in S$ is a *deadlock* if it is not final and has no outgoing transitions. An LTS is *deadlock-free* if no reachable state is a deadlock. Given an LTS $\mathcal{S} = (S, \Gamma, T, s_0, F)$ and a state $s \in S$, we use $\mathcal{S}_s$ to denote the LTS obtained by replacing s_0 with s as the initial state: (S, Γ, T, s, F).

3.1 Global Communicating Labeled Transition Systems (GCLTS)

We use LTS over the synchronous alphabet Γ_{sync} to model communication protocols from a global perspective. We impose three more conditions on the class of LTSs we consider: that final states do not contain outgoing transitions, that multiple outgoing transitions from a state share a sender, and that the LTS is deadlock-free.

Definition 3.1 (Global communicating LTS). An LTS $\mathcal{S} = (S, \Gamma_{sync}, T, s_0, F)$ is a *global communicating labeled transition system* (GCLTS) if the following conditions hold:

(1) *sink-finality*: for every final state $s \in F$, there does not exist $l \in \Gamma_{sync}$ and $s' \in S$ with $s \xrightarrow{l} s' \in T$;
(2) *sender-driven choice*: for all states $s, s_1, s_2 \in S$ and $l_1, l_2 \in \Gamma_{sync}$ such that $s \xrightarrow{l_i} s_i \in T$ for $i \in \{1, 2\}$, there is a participant $\mathsf{p} \in \mathcal{P}$ who is the sender for both, i.e. $\texttt{split}(l_i) \in \Sigma_{\mathsf{p},!}$ for $i \in \{1, 2\}$, and furthermore $l_1 = l_2 \implies s_1 = s_2$;
(3) *deadlock freedom*: $\mathcal{S}$ is deadlock-free.

Condition (1) is ubiquitous in the domain of multiparty session types and was also shown to require special treatment in the literature on high-level message sequence charts [18].

Condition (2) is a generalisation of most multiparty session types fragments, which require not only a dedicated sender but also a dedicated receiver. This more restrictive condition is called *directed choice*. In contrast, *mixed choice* lifts all restrictions on choice, and amounts to only requiring determinism. Lohrey [59] showed that implementability is undecidable for HMSCs satisfying determinism and Condition (3). Stutz [79] showed that implementability remains undecidable for mixed choice global multiparty session types satisfying determinism and Conditions (1) and (3). Sender-driven choice thus represents a good middle ground, allowing to express interesting communication patterns while retaining decidability of implementability.

Condition (3) simply requires that protocols do not specify deadlocking behaviors.

In the remainder of the paper we refer to a GCLTS simply as a *protocol*.

Restricting Protocols to Participants. From a protocol $\mathcal{S}$, we can define a local protocol for each participant p via domain restriction to Σ_{p}. Formally, given a protocol $\mathcal{S} = (S, \Gamma_{sync}, T, s_0, F)$, we define $\mathcal{S}_{\mathsf{p}} \coloneqq (S, \Gamma_{\mathsf{p}} \uplus \{\varepsilon\}, T_{\mathsf{p}}, s_0, F)$ where $T_{\mathsf{p}} \coloneqq \{s \xrightarrow{l\Downarrow_{\Gamma_{\mathsf{p}}}} s' \mid s \xrightarrow{l} s' \in T\}$ for a participant $\mathsf{p} \in \mathcal{P}$.

Asynchronous Protocol Semantics. Note that a protocol is specified using the synchronous alphabet Γ_{sync}. To define the *asynchronous* semantics of a protocol $\mathcal{S}$ we first map finite and infinite words of $\mathcal{S}$ onto their asynchronous counterpart using $\texttt{split}$, thus obtaining a set of asynchronous words

in which matching send and receive events are adjacent to each other. In an asynchronous, FIFO network, two events are independent if they are not related by the *happened-before* relation [53]. For example, any two send events from distinct senders are independent. Consequently, two words are *indistinguishable* if any asynchronous, FIFO implementation that recognizes one word must recognize the other, e.g. $w \cdot \mathsf{p} \triangleright \mathsf{q}!m \cdot \mathsf{r} \triangleright \mathsf{s}!m' \cdot u$ and $w \cdot \mathsf{r} \triangleright \mathsf{s}!m' \cdot \mathsf{p} \triangleright \mathsf{q}!m \cdot u$, where $\mathsf{p} \neq \mathsf{r}$. We define our protocol semantics as the set of *channel-compliant* [60] words that are closed under this notion of indistinguishability. Channel compliance characterizes words that respect FIFO order, i.e. receive events appear after their matching send event, and the order of receive events follows that of send events in each channel.

Definition 3.2 (Channel compliance). Let $w \in \Sigma^{\infty}_{async}$. We say that w is *channel-compliant* if for all prefixes $w' \leq w$, for all $\mathsf{p} \neq \mathsf{q} \in \mathcal{P}$, $\mathcal{V}(w'\Downarrow_{\mathsf{q}\triangleleft\mathsf{p}?\_}) \leq \mathcal{V}(w'\Downarrow_{\mathsf{p}\triangleright\mathsf{q}!\_})$.

The asynchronous semantics of a protocol is defined as follows:

$$\begin{aligned} C^{\sim}(\mathcal{S}) = \{w' \in \Sigma^{*}_{async} \mid \exists w \in \Sigma^{*}_{async}.\, w \in \mathtt{split}(\mathcal{L}(\mathcal{S})) \wedge w' \text{ is channel-compliant} \\ \wedge \forall \mathsf{p} \in \mathcal{P}.\, w'\Downarrow_{\Sigma_{\mathsf{p}}} = w\Downarrow_{\Sigma_{\mathsf{p}}}\} \\ \cup \{w' \in \Sigma^{\omega}_{async} \mid \exists w \in \Sigma^{\omega}_{async}.\, w \in \mathtt{split}(\mathcal{L}(\mathcal{S})) \wedge \forall v' \leq w'.\, \exists u, u' \in \Sigma^{*}_{async}. \\ v' \cdot u' \text{ is channel-compliant} \ \wedge\ u \leq w \ \wedge \forall \mathsf{p} \in \mathcal{P}.\, (v' \cdot u')\Downarrow_{\Sigma_{\mathsf{p}}} = u\Downarrow_{\Sigma_{\mathsf{p}}}\} \ . \end{aligned}$$

Membership of infinite words is defined in terms of their prefixes: every prefix v' must be channel-compliant, and moreover extensible to a word $v'u'$ that is indistinguishable from another prefix already in the language. Since we do not make any fairness assumptions on scheduling, the semantics of infinite words includes traces such as $(\mathsf{p} \triangleright \mathsf{q}!m)^{\omega}$ for the protocol $(\mathsf{p} \triangleright \mathsf{q}!m.\mathsf{q} \triangleleft \mathsf{p}?m)^{\omega}$, where only the sender is scheduled. Membership of finite words follows standard MSC semantics.

In the remainder of the paper, we overload notation and use $\mathcal{L}(\mathcal{S})$ to denote $C^{\sim}(\mathcal{S})$.

Communicating LTS. We use communicating LTS to model the local behaviors of participants: $\mathcal{T} = \{\!\{T_{\mathsf{p}}\}\!\}_{\mathsf{p}\in\mathcal{P}}$ is a *communicating labeled transition system* (CLTS) over $\mathcal{P}$ and $\mathcal{V}$ if T_{p} is a deterministic LTS over Σ_{p} for every $\mathsf{p} \in \mathcal{P}$, denoted by $(Q_{\mathsf{p}}, \Sigma_{\mathsf{p}}, \delta_{\mathsf{p}}, q_{0,\mathsf{p}}, F_{\mathsf{p}})$. Let $\prod_{\mathsf{p}\in\mathcal{P}} Q_{\mathsf{p}}$ denote the set of global states and $\mathsf{Chan} = \{(\mathsf{p},\mathsf{q}) \mid \mathsf{p},\mathsf{q} \in \mathcal{P}, \mathsf{p} \neq \mathsf{q}\}$ denote the set of channels. A *configuration* of $\mathcal{A}$ is a pair $(\vec{s}, \xi)$, where $\vec{s}$ is a global state and $\xi : \mathsf{Chan} \to \mathcal{V}^*$ is a mapping from each channel to a sequence of messages. We use $\vec{s}_{\mathsf{p}}$ to denote the state of p in $\vec{s}$. The CLTS transition relation, denoted $\rightarrow$, is defined as follows.

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

In the initial configuration $(\vec{s}_0, \xi_0)$, each participant's state in $\vec{s}_0$ is the initial state $q_{0,\mathsf{p}}$ of A_{p}, and ξ_0 maps each channel to ε. A configuration $(\vec{s}, \xi)$ is *final* iff $\vec{s}_{\mathsf{p}}$ is final for every 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 $\mathcal{L}(\mathcal{T})$ of the CLTS $\mathcal{T}$ is defined as the set of maximal traces. A configuration $(\vec{s}, \xi)$ is a *deadlock* if it is not final and has no outgoing transitions. A CLTS is *deadlock-free* if no reachable configuration is a deadlock.

Observe that in a CLTS, send transitions are always enabled, whereas receive transitions are only enabled if the message exists at the head of its corresponding channel. Communicating state machines [8] are a special case of CLTS where the LTS for each participant $\mathsf{p} \in \mathcal{P}$ is a deterministic finite state machine. Note that CLTS describe asynchronous communication with message channels

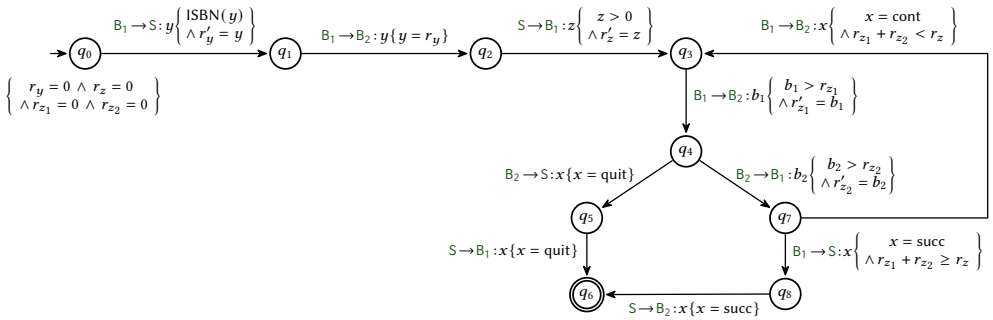

Fig. 7. The two-bidder protocol from Fig. 1 as a symbolic protocol with registers r_z, r_y, r_{z_1}, and r_{z_2}.

of unbounded size. Thus, they differ from Zielonka's *asynchronous automata* [90], which actually describe *synchronously communicating systems* [67]. We refer the reader to [26] for further details.

Finally, we define protocol implementability.

Definition 3.3 (Protocol Implementability). A protocol $\mathcal{S}$ is *implementable* if there exists a CLTS $\{\!|T_{\mathsf{p}}|\!\}_{\mathsf{p}\in\mathcal{P}}$ such that the following two properties hold: (i) *protocol fidelity*: $\mathcal{L}(\{\!|T_{\mathsf{p}}|\!\}_{\mathsf{p}\in\mathcal{P}}) = \mathcal{L}(\mathcal{S})$, and (ii) *deadlock freedom*: $\{\!|T_{\mathsf{p}}|\!\}_{\mathsf{p}\in\mathcal{P}}$ is deadlock-free. We say that $\{\!|T_{\mathsf{p}}|\!\}_{\mathsf{p}\in\mathcal{P}}$ implements $\mathcal{S}$.

A notion of implementability that relaxes language equality to language inclusion has been studied as protocol refinement [55]. Alternatively, one can expand the set of protocol behaviors to include deadlocking behaviors, resulting in a notion of implementability that replaces language equality with prefix set equality, and waives the requirement of deadlock freedom.

3.2 Symbolic Protocols with Dependent Refinements

We now introduce our model for finitely representing infinite state protocols. We refer to these representations simply as *symbolic protocols*. Figure 7 shows the two-bidder protocol from Fig. 1 expressed as a symbolic protocol.

The formal definition of this class of symbolic protocols is given below. In this definition, we assume a fixed but unspecified first-order background theory of message values (e.g. linear integer arithmetic). We assume standard syntax and semantics of first-order formulas and denote by $\mathcal{F}$ the set of first-order formulas with free variables drawn from an infinite set X. We assume that these variables are interpreted over the set of message values $\mathcal{V}$. For a valuation $\rho \in X \to \mathcal{V}$ and $\varphi \in \mathcal{F}(X)$, we write $\rho \models \phi$ to indicate that φ evaluates to true under ρ in the underlying theory.

Definition 3.4 (Symbolic protocol). A symbolic protocol is a tuple $\mathbb{S} = (S, R, \Delta, s_0, \rho_0, F)$ where

- S is a finite set of control states,
- R is a finite set of register variables,
- $\Delta \subseteq S \times \mathcal{P} \times X \times \mathcal{P} \times \mathcal{F} \times S$ is a finite set that consists of symbolic transitions of the form $s \xrightarrow{\mathsf{p}\to\mathsf{q}:x\{\varphi\}} s'$ where the formula φ with free variables $R \uplus R' \uplus \{x\}$ expresses a transition constraint that relates the old and new register values (R and R'), and the sent value x,
- $s_0 \in S$ is the initial control state,
- $\rho_0 : R \to \mathcal{V}$ is the initial register assignment, and
- $F \subseteq S$ is a set of final states.

To streamline our definition, we choose not to separate register update expressions from predicates describing the communication. Rather, we specify everything together in a single formula φ,

that can only talk about the current value being communicated, and the register values in the pre- and post-state. Thus, φ can describe values that are communicated between participants, in addition to register assignments and updates. For example, $\mathsf{p} \to \mathsf{q} : x\{even(x) \wedge r_1' = r_1 + 2 \wedge r_2' = x\}$ describes p sending q an even number x, incrementing the value of register r_1 by 2, and storing the value of x in register r_2. We formally specify the two-bidder protocol from Fig. 1 as a symbolic protocol in Fig. 7 for demonstration purposes; note that the transition predicate $\mathsf{ISBN}(y)$ from q_1 to q_2 is replaced with an equality. For readability and conciseness, we employ the following conventions from now on. We treat communication variables as registers that are automatically assigned the communicated value, e.g. $\mathsf{S} \to \mathsf{B}_1 : z\{z > 0\}$ should be understood as $\mathsf{S} \to \mathsf{B}_1 : x\{x > 0 \wedge z' = x\}$ for some fresh x. Furthermore, if the communicated value is a constant c and there is no need to store this value, we inline it and write $\mathsf{S} \to \mathsf{B}_2 : \mathsf{succ}\{\top\}$ instead of $\mathsf{S} \to \mathsf{B}_2 : x\{x = \mathsf{succ}\}$. We may omit the condition $\top$, turning $\mathsf{S} \to \mathsf{B}_2 : \mathsf{succ}\{\top\}$ into $\mathsf{S} \to \mathsf{B}_2 : \mathsf{succ}$.

Symbolic protocols are specification-wise similar to symbolic register automata [19], but allow more general patterns of register manipulation and do not a priori require formulas to come from an effective Boolean algebra. Symbolic protocols can be seen as a finite description of an infinite-state LTS, whose concrete states consist of a control state along with an assignment for the register variables R. Transitions are concrete communication events that optionally modify register values. We formally define the concretization of a symbolic protocol below.

Definition 3.5 (Concretization of symbolic protocols). For a symbolic protocol $\mathbb{S} = (S, R, \Delta, s_0, \rho_0, F)$, let $\mathcal{S}_{\mathbb{S}}$ denote its concrete protocol. The set of states of $\mathcal{S}_{\mathbb{S}}$ is $S \times (R \to \mathcal{V})$.

Transitions in $\mathcal{S}_{\mathbb{S}}$ are defined as follows:

$$\frac{s_1 \xrightarrow{\mathsf{p}\to\mathsf{q}:x\{\varphi\}} s_2 \in \Delta \qquad \rho_1\rho_2'[x \mapsto v] \models \varphi}{(s_1, \rho_1) \xrightarrow{\mathsf{p}\to\mathsf{q}:v} (s_2, \rho_2)}$$

Intuitively, the rule says that a symbolic transition from s_1 to s_2 can be instantiated to one from (s_1, ρ_1) to (s_2, ρ_2) on value v when v together with the register assignments in the pre- and post-state satisfy the transition constraint φ. Here, we use juxtaposition $\rho_1\rho_2'$ of register assignments to express their disjoint union. The assignment ρ_2' is obtained from ρ_2 by replacing registers r in the domain with their primed version in R'. The initial state is defined as (s_0, ρ_0). A state (s, ρ) in $\mathcal{S}_{\mathbb{S}}$ is final when $s \in F$.

Thus, the concrete protocol $\mathcal{S}_{\mathbb{S}}$ is a protocol over the alphabet Γ_{sync}. The language of a symbolic protocol $\mathbb{S}$ is defined as the language of its concretization $\mathcal{S}_{\mathbb{S}}$. Consequently, a symbolic protocol is implementable if its concretization is implementable.

4 Characterizing Protocol Implementability

We motivate our precise characterization of protocol implementability through examples of non-implementable protocols, and show that seemingly disparate sources of non-implementability share a unified semantic explanation. Recall the protocol $\mathbb{S}_1$ from §2 with a receiver violation, depicted in Fig. 5. The infinite-state LTS $\mathbb{S}_1$ contains the two concrete run prefixes depicted in Fig. 8, where the values of x, y are 2, 3 and 1, 3 respectively.

Inspecting $\mathbb{S}_1$'s specification reveals that the protocol expects r to receive messages from p and q in a different order depending on the branch that q chooses to follow. However, this expectation is unreasonable in a distributed setting. Between the two concrete runs, r's partial view of the protocol's behavior is the same: r receives a value 3 from p, yet r is expected to receive in p, q order in one run, and receive in q, p order in the other.

Recall the protocol $\mathbb{S}_2$ from §2 with a sender violation, depicted in Fig. 6. Again inspecting $\mathbb{S}_2$'s

specification, the branching structure imposes the expectation that on the top branch, p should send q an o message, whereas on the bottom branch, p should immediately terminate. The two concrete runs in Fig. 9 and Fig. 10 again demonstrate that this expectation is unreasonable: p receives the value 3 from q in both runs, but in one run is expected to send a message, whereas in the other is expected to terminate.

The non-implementability in the examples above can be attributed to insufficient local information about protocol control flow. This source of non-implementability is inherent to the expressive power of branching choice in protocol specifications, and is present even in finite protocols with more restricted choice constructs. While most existing works soundly detect insufficient local information through conservative projection algorithms [14, 45, 75, 81], Li et al. [56] give a complete characterization. To check implementability, they first obtain a candidate implementation by restricting the global protocol onto each participant's alphabet, and then determinizing the resulting finite state automaton. Then, they check sound and complete conditions directly on the states of the candidate implementation.

Our first observation towards a precise characterization is that implementability can be checked directly on the global protocol specification, without synthesizing a candidate implementation upfront. This is especially important in the general case, when synthesizing a candidate implementation is itself challenging and not always possible. Our analysis of the protocols above shows that non-implementability can be blamed solely on the existence of certain states in the concrete LTS represented by the global protocol. In fact, we show in §5 that the implementability check for global types by Li et al. [56] can be made more efficient by forgoing the synthesis step.

Let us now turn our attention to a different source of non-implementability that is unique to the setting of dependent data refinements. Consider the following pair of symbolic protocols $\mathbb{S}_3$ and $\mathbb{S}'_3$, depicted in Fig. 11 and Fig. 12.

Non-implementability is again caused by insufficient local information, but this time with respect to message data rather than control flow: in fact, no branching choice appears in this pair of simple protocols. The problem instead arises in the fact that in both $\mathbb{S}_3$ and $\mathbb{S}'_3$, r does not know the value of x. While an implementation for r could produce a subset of $\mathbb{S}_3$'s behaviors (e.g. by sending z such that $z > y$), or produce a superset of $\mathbb{S}_3$'s behaviors (e.g. by sending all values for z), no implementation can produce exactly the specified behaviors, as required by protocol fidelity. Zhou et al. [89] address partial information of protocol variables by syntactically classifying whether a variable is known or unknown to a participant, and annotating the variables accordingly in the typing context: a variable is known to its sender and receiver, and unknown to all other participants. However, this syntactic analysis is itself insufficient, as demonstrated by these examples: both protocols yield the same classification of variables per participant, yet one is implementable and the other is not.

We instead turn to concrete runs of $\mathbb{S}_3$ to find the source of non-implementability. Let us consider the concrete runs of $\mathbb{S}_3$ depicted in Fig. 13, where the values of x, y are 2, 4 and 3, 4 respectively.

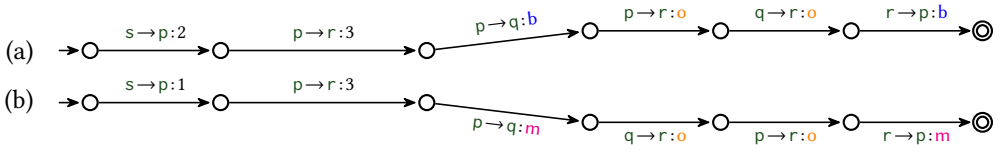

Fig. 8. Two concrete runs of $\mathbb{S}_1$ (Fig. 5): (a) with $x = 2$ and $y = 3$ and (b) with $x = 1$ and $y = 3$.

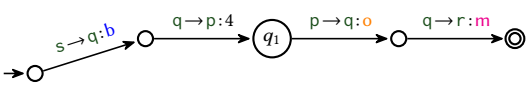

Fig. 9. A concrete run of $\mathbb{S}_2$ (Fig. 6) with s choosing the top branch.

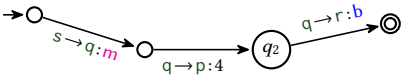

Fig. 10. A concrete run of $\mathbb{S}_2$ (Fig. 6) with s choosing the bottom branch and $x_2 = 4$.

In this pair of runs, r observes the same behaviors, namely receiving the value 4 from q. While $\mathbb{S}_3$ also permits r to send 4 to p in the first run, sending 3 to p in the second run constitutes a violation to the refinement predicate $z > x$, i.e. $3 > 3$ is false. Again, this presents a problem because the two run prefixes are indistinguishable to r. Observe that in this example, non-implementability can again be blamed solely on the existence of states in the global protocol.

We formalize a participant's local information about the protocol using two variations on the standard notion of reachability. Let $\mathcal{S} = (S, \Gamma_{sync}, T, s_0, F)$ be a protocol and let $\mathsf{p} \in \mathcal{P}$ be a participant. The standard notion defines s' as reachable from s in $\mathcal{S}$ on $w \in \Gamma_{sync}^*$, denoted $s \xrightarrow{w}{}^* s'$, when there exists a sequence of transitions $s_1 \xrightarrow{l_1} s_2 \dots s_{n-1} \xrightarrow{l_{n-1}} s_n$, such that $s_1 = s$, $s_n = s'$, $l_1 \dots l_{n-1} = w$ and for each $1 \le i < n$, it holds that $s_i \xrightarrow{l_i} s_{i+1} \in T$. We first define a notion of reachability that restricts the transitions to only the actions observable by a single participant.

Participant-based Reachability. We say that $s \in S$ is *reachable for* p *on* $u \in \Gamma_{\mathsf{p}}^*$ when there exists $w \in \Gamma_{sync}^*$ such that $s_0 \xrightarrow{w}{}^* s \in T$ and $w{\Downarrow_{\Gamma_{\mathsf{p}}}} = u$, which we denote $s_0 \overset{u}{\underset{\mathsf{p}}{\Longrightarrow}}{}^* s$. We characterize *simultaneously reachable* pairs of states for each participant using the notion of participant-based reachability.

Simultaneous Reachability. We say that $s_1, s_2 \in S$ are simultaneously reachable for participant p on $u \in \Gamma_{\mathsf{p}}^*$, denoted $s_0 \overset{u}{\underset{\mathsf{p}}{\Longrightarrow}}{}^* s_1, s_2$, if there exist $w_1, w_2 \in \Gamma_{sync}^*$ such that $s_0 \xrightarrow{w_1}{}^* s_1 \in T, s_0 \xrightarrow{w_2}{}^* s_2 \in T$ and $w_1{\Downarrow_{\Gamma_{\mathsf{p}}}} = w_2{\Downarrow_{\Gamma_{\mathsf{p}}}} = u$. Simultaneous reachability captures the notion of *locally indistinguishable* states: to a participant, two states are locally indistinguishable if they are simultaneously reachable.

Send Coherence requires that any message that can be sent from a state can also be sent from all other states that are locally indistinguishable to the sender.

Definition 4.1 (Send Coherence). A protocol $\mathcal{S} = (S, \Gamma_{sync}, T, s_0, F)$ satisfies Send Coherence (SC) when for every $s_1 \xrightarrow{\mathsf{p}\to\mathsf{q}:m} s_2 \in T, s_1' \in S$:

$$(\exists u \in \Gamma_{\mathsf{p}}^*.\ s_0 \overset{u}{\underset{\mathsf{p}}{\Longrightarrow}}{}^* s_1, s_1') \implies (\exists s_2' \in S.\ s_1' \overset{\mathsf{p}\to\mathsf{q}:m}{\underset{\mathsf{p}}{\Longrightarrow}}{}^* s_2') \ .$$

Receive Coherence, on the other hand, requires that no message which can be received from a state can be received from any other state that is locally indistinguishable to the receiver.

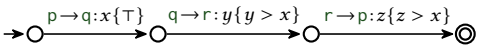

Fig. 11. $\mathbb{S}_3$: A non-implementable protocol with dependent refinements.

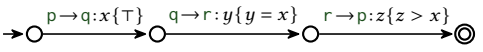

Fig. 12. $\mathbb{S}_3'$: An implementable protocol with dependent refinements.

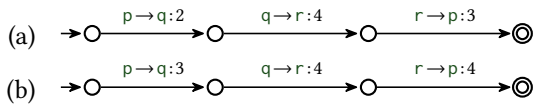

Fig. 13. Two concrete runs of $\mathbb{S}_3$ (Fig. 11): (a) with $x = 2, y = 4, z = 3$ and (b) with $x = 3, y = 4, z = 4$.

Definition 4.2 (Receive Coherence). A protocol $\mathcal{S} = (S, \Gamma_{sync}, T, s_0, F)$ satisfies Receive Coherence (RC) when for every $s_1 \xrightarrow{\mathsf{p}\to\mathsf{q}:m} s_2, s_1' \xrightarrow{\mathsf{r}\to\mathsf{q}:m} s_2' \in T$:

$$(\mathsf{r} \neq \mathsf{p} \wedge \exists u \in \Gamma_{\mathsf{q}}^*.\ s_0 \overset{u}{\underset{\mathsf{q}}{\Longrightarrow}}{}^* s_1, s_1') \implies \forall w \in \mathrm{pref}(\mathcal{L}(\mathcal{S}_{s_2'})).\ w{\Downarrow_{\Sigma_{\mathsf{q}}}} \neq \varepsilon \vee \mathcal{V}(w{\Downarrow_{\mathsf{p}\triangleright\mathsf{q}!\_}}) \neq \mathcal{V}(w{\Downarrow_{\mathsf{q}\triangleleft\mathsf{p}?\_}})\cdot m)\ .$$

No Mixed Choice requires that roles cannot equivocate between sending and receiving in two locally indistinguishable states.

Definition 4.3 (No Mixed Choice). A protocol $\mathcal{S} = (S, \Gamma_{sync}, T, s_0, F)$ satisfies No Mixed Choice (NMC) when for every $s_1 \xrightarrow{\mathsf{p}\to\mathsf{q}:m} s_2, s_1' \xrightarrow{\mathsf{r}\to\mathsf{p}:m} s_2' \in T$: $(\exists u \in \Gamma_{\mathsf{p}}^*.\ s_0 \overset{u}{\underset{\mathsf{p}}{\Longrightarrow}}{}^* s_1, s_1') \implies \bot$.

Our semantic characterization of protocol implementability is the conjunction of the above three conditions. In contrast to the syntactic analysis in [89], our semantic approach is sound and complete. In contrast to the sound and complete approach in [56], our implementability conditions do not rely on synthesizing an implementation upfront.

Definition 4.4 (Coherence Conditions). A protocol satisfies Coherence Conditions (CC) when it satisfies Send Coherence, Receive Coherence and No Mixed Choice.

The preciseness of *CC* is stated as follows.

Theorem 4.5. *Let $\mathcal{S}$ be a protocol. Then, $\mathcal{S}$ is implementable if and only if it satisfies CC.*

In the next two sections, we illustrate the key steps for proving Theorem 4.5. We refer the reader to the extended version [58] for the complete proofs.

4.1 Soundness

Soundness requires us to show that if a protocol satisfies *CC*, then it is implementable. We begin by echoing the observation made in several prior works [1, 56, 78] that for any global protocol, there exists a *canonical* implementation consisting of one local implementation per participant. We formally define what it means for an implementation to be canonical in our setting below.

Definition 4.6 (Canonical implementations). We say a CLTS $\{\!|T_{\mathsf{p}}|\!\}_{\mathsf{p}\in\mathcal{P}}$ is a *canonical implementation* for a protocol $\mathcal{S} = (S, \Gamma_{sync}, T, s_0, F)$ if for every $\mathsf{p} \in \mathcal{P}$, T_{p} satisfies:
(i) $\forall w \in \Sigma_{\mathsf{p}}^*.\ w \in \mathcal{L}(T_{\mathsf{p}}) \Leftrightarrow w \in \mathcal{L}(\mathcal{S}){\Downarrow_{\Sigma_{\mathsf{p}}}}$, and (ii) $\mathrm{pref}(\mathcal{L}(T_{\mathsf{p}})) = \mathrm{pref}(\mathcal{L}(\mathcal{S}){\Downarrow_{\Sigma_{\mathsf{p}}}})$.

We first prove that following fact about canonical implementations of protocols satisfying NMC, which states that the canonical implementations themselves do not exhibit mixed choice.

Lemma 4.7 (No Mixed Choice). *Let $\mathcal{S}$ be a protocol satisfying NMC (Definition 4.3) and let $\{\!|T_{\mathsf{p}}|\!\}_{\mathsf{p}\in\mathcal{P}}$ be a canonical implementation for $\mathcal{S}$. Let $wx_1, wx_2 \in \mathrm{pref}(\mathcal{L}(T_{\mathsf{p}}))$ with $x_1 \neq x_2$ for some $\mathsf{p} \in \mathcal{P}$. Then, $x_1 \in \Sigma_!$ iff $x_2 \in \Sigma_!$.*

We choose the canonical implementation as our existential witness to show that any protocol satisfying *CC* is implementable. By the definition of implementability (Definition 3.3), soundness

amounts to showing the following three conditions:
(a) $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\{\!|T_\mathsf{p}|\!\}_{\mathsf{p}\in\mathcal{P}})$, (b) $\mathcal{L}(\{\!|T_\mathsf{p}|\!\}_{\mathsf{p}\in\mathcal{P}}) \subseteq \mathcal{L}(\mathcal{S})$, and (c) $\{\!|T_\mathsf{p}|\!\}_{\mathsf{p}\in\mathcal{P}}$ is deadlock-free.

Condition (a) states that any canonical implementation recognizes at least the global protocol behaviors. This fact can be shown for any LTS and canonical CLTS, and does not rely on assumptions about determinism or sender-drivenness, nor assumptions about the LTS satisfying *CC*.

Lemma 4.8 (Canonical implementation language contains protocol language). *Let $\mathcal{S}$ be an LTS and let $\{\!|T_\mathsf{p}|\!\}_{\mathsf{p}\in\mathcal{P}}$ be a canonical implementation for $\mathcal{S}$. Then, $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\{\!|T_\mathsf{p}|\!\}_{\mathsf{p}\in\mathcal{P}})$.*

Condition (b), on the other hand, states that any behavior recognized by the canonical implementation is a global protocol behavior, in other words, that the canonical CLTS does not add behaviors. This is only true for protocols that satisfy *CC*.

Furthermore, the acceptance condition for infinite words in $\mathcal{L}(\mathcal{S})$ differs from that in $\{\!|T_\mathsf{p}|\!\}_{\mathsf{p}\in\mathcal{P}}$: the latter accepts all infinite traces, whereas the former requires to show that an infinite word w satisfies $w \preceq^{\omega}_{\sim} w'$ for some other infinite word $w' \in \mathcal{L}(\mathcal{S})$. Therefore, showing prefix set inclusion is not sufficient, and we must reason about the finite and infinite case separately.

Lemma 4.9 (Global protocol language contains canonical implementation language). *Let $\mathcal{S}$ be a protocol satisfying* CC *and let $\{\!|T_\mathsf{p}|\!\}_{\mathsf{p}\in\mathcal{P}}$ be a canonical implementation for $\mathcal{S}$ such that for all $w \in \Sigma^*_{async}$, if w is a trace of $\{\!|T_\mathsf{p}|\!\}_{\mathsf{p}\in\mathcal{P}}$, then $I(w) \neq \emptyset$. Then, $\mathcal{L}(\{\!|T_\mathsf{p}|\!\}_{\mathsf{p}\in\mathcal{P}}) \subseteq \mathcal{L}(\mathcal{S})$.*

Towards these ends, we adapt the key intermediate lemma from [56] to our setting, and show the inductive invariant that every trace in the canonical implementation of a protocol satisfying *CC* satisfies *intersection set non-emptiness*. Note that although our intermediate lemma statements are similar to those in [56] in structure, [56] reasons about a particular implementation, namely the subset construction obtained from the global type, whereas our proofs reason about any canonical implementation of a global protocol that satisfies *CC*. As a result, the proof arguments differ significantly.

We adapt the relevant definitions to our setting below.

Definition 4.10 (LTS intersection sets). Let $\mathcal{S}$ be an LTS. Let p be a participant and $w \in \Sigma^*_{async}$ be a word. We define the set of possible runs $\mathrm{R}^{\mathcal{S}}_{\mathsf{p}}(w)$ as all maximal runs of $\mathcal{S}$ that are consistent with p's local view of w:

$$\mathrm{R}^{\mathcal{S}}_{\mathsf{p}}(w) \coloneqq \{\rho \text{ is a maximal run of } \mathcal{S} \mid w{\Downarrow_{\Sigma_\mathsf{p}}} \leq \mathtt{split}(\mathtt{trace}(\rho)){\Downarrow_{\Sigma_\mathsf{p}}}\} \ .$$

We denote the intersection of the possible run sets for all participants as $I^{\mathcal{S}}(w) \coloneqq \bigcap_{\mathsf{p}\in\mathcal{P}} \mathrm{R}^{\mathcal{S}}_{\mathsf{p}}(w)$.

Definition 4.11 (Unique splitting of a possible run). Let $\mathcal{S}$ be an LTS, p a participant, and $w \in \Sigma^*_{async}$ a word. Let ρ be a run in $\mathrm{R}^{\mathcal{S}}_{\mathsf{p}}(w)$. We define the longest prefix of ρ matching w:

$$\alpha' \coloneqq \max\{\rho' \mid \rho' \leq \rho \ \wedge \ \mathtt{split}(\mathtt{trace}(\rho')){\Downarrow_{\Sigma_\mathsf{p}}} \leq w{\Downarrow_{\Sigma_\mathsf{p}}}\} \ .$$

If $\alpha' \neq \rho$, we can split ρ into $\rho = \alpha \cdot s \xrightarrow{l} s' \cdot \beta$ where $\alpha' = \alpha \cdot s.$, which we call the unique splitting of ρ for p matching w. Uniqueness follows from the maximality of α'.

For example, the unique splitting of $\rho = s_1 \xrightarrow{\mathsf{p}\to\mathsf{q}:\mathsf{m}} s_2 \xrightarrow{\mathsf{r}\to\mathsf{q}:\mathsf{b}_1} s_3 \xrightarrow{\mathsf{r}\to\mathsf{q}:\mathsf{b}_2} s_4 \xrightarrow{\mathsf{q}\to\mathsf{p}:\mathsf{o}} s_5$ for p matching $w = \mathsf{r} \triangleright \mathsf{q}!\mathsf{b}_1.\,\mathsf{p} \triangleright \mathsf{q}!\mathsf{m}$ is $\alpha \cdot s_3 \xrightarrow{\mathsf{r}\to\mathsf{q}:\mathsf{b}_2} s_4 \cdot \beta$, where $\alpha = s_1 \xrightarrow{\mathsf{p}\to\mathsf{q}:\mathsf{m}} s_2 \xrightarrow{\mathsf{r}\to\mathsf{q}:\mathsf{b}_1} s_3$ and $\beta = s_4 \xrightarrow{\mathsf{q}\to\mathsf{p}:\mathsf{o}} s_5$.

Our intersection non-emptiness inductive invariant is stated below. The proof proceeds by induction on the length of a prefix w of the canonical implementation, and case splits based on whether w is extended by a send or receive action. Lemma 4.14 and Lemma 4.13 provide a characterization for each case respectively.

Lemma 4.12 (Intersection set non-emptiness). *Let $\mathcal{S}$ be a protocol satisfying* CC*, and let $\{\!|T_{\mathsf{p}}|\!\}_{\mathsf{p}\in\mathcal{P}}$ be a canonical implementation for $\mathcal{S}$. Then, for every trace $w \in \Sigma^*_{async}$ of $\{\!|T_{\mathsf{p}}|\!\}_{\mathsf{p}\in\mathcal{P}}$, it holds that $I(w) \neq \emptyset$.*

Lemma 4.13 (Receive events do not shrink intersection sets). *Let $\mathcal{S}$ be a protocol satisfying* CC*, and let $\{\!|T_{\mathsf{p}}|\!\}_{\mathsf{p}\in\mathcal{P}}$ be a canonical implementation for $\mathcal{S}$. Let wx be a trace of $\{\!|T_{\mathsf{p}}|\!\}_{\mathsf{p}\in\mathcal{P}}$ such that $x \in \Sigma_?$. Then, $I(w) = I(wx)$.*

Lemma 4.14 (Send events preserve run prefixes). *Let $\mathcal{S}$ be a protocol satisfying* CC *and $\{\!|T_{\mathsf{p}}|\!\}_{\mathsf{p}\in\mathcal{P}}$ be a canonical implementation for $\mathcal{S}$. Let wx be a trace of $\{\!|T_{\mathsf{p}}|\!\}_{\mathsf{p}\in\mathcal{P}}$ such that $x \in \Sigma_{\mathsf{p},!}$ for some $\mathsf{p} \in \mathcal{P}$. Let ρ be a run in $I(w)$, and $\alpha \cdot s_{pre} \xrightarrow{l} s_{post} \cdot \beta$ be the unique splitting of ρ for p with respect to w. Then, there exists a run ρ' in $I(wx)$ such that $\alpha \cdot s_{pre} \leq \rho'$.*

Finally, we show that protocols that satisfy *CC* and intersection set non-emptiness have deadlock-free canonical implementations. The proof follows immediately from the following lemma and the fact that CLTS are deterministic, and is thus omitted.

Lemma 4.15 (Channel compliance and intersection set non-emptiness implies prefix). *Let $\mathcal{S} = (S, \Gamma_{sync}, T, s_0, F)$ be a protocol and let $w \in \Sigma^*_{async}$ be a word such that (i) w is channel-compliant, and (ii) $I(w) \neq \emptyset$. Then, $w \in \mathrm{pref}(\mathcal{L}(\mathcal{S}))$.*

Lemma 4.16 (Canonical implementation deadlock freedom). *Let $\mathcal{S} = (S, \Gamma_{sync}, T, s_0, F)$ be a protocol satisfying* CC *and let $\{\!|T_{\mathsf{p}}|\!\}_{\mathsf{p}\in\mathcal{P}}$ be a canonical implementation for $\mathcal{S}$ such that for all $w \in \Sigma^*_{async}$, if w is a trace of $\{\!|T_{\mathsf{p}}|\!\}_{\mathsf{p}\in\mathcal{P}}$, then $I(w) \neq \emptyset$. Then, $\{\!|T_{\mathsf{p}}|\!\}_{\mathsf{p}\in\mathcal{P}}$ is deadlock-free.*

Soundness thus follows from the three conditions above.

Lemma 4.17 (Soundness of *CC*). *Let $\mathcal{S}$ be a protocol. If $\mathcal{S}$ satisfies* CC*, then $\mathcal{S}$ is implementable.*

4.2 Completeness

Completeness requires us to show that if a protocol is implementable, then it satisfies *CC*. We prove completeness by modus tollens, and assume that a protocol $\mathcal{S}$ does not satisfy *CC*. We negate SC, RC and NMC in turn: from the negation of SC we obtain a simultaneously reachable pair of states in $\mathcal{S}$ such that a send event that is enabled in one is never enabled from the other. From the negation of RC there exists a simultaneously reachable pair of states in $\mathcal{S}$ such that a receive event that is enabled in one is also enabled in the other. From the negation of NMC we obtain a simultaneously reachable pair of transitions where a participant is the sender in one and the receiver in the other. We assume an arbitrary CLTS that implements $\mathcal{S}$, and use these witnesses to show that this CLTS must recognize a trace that is not a prefix in $\mathcal{L}(\mathcal{S})$, thereby either violating protocol fidelity or deadlock freedom.

Lemma 4.18 (Completeness). *Let $\mathcal{S}$ be a protocol. If $\mathcal{S}$ is implementable, then $\mathcal{S}$ satisfies* CC*.*

An immediate consequence of the soundness and completeness of *CC* is the following fact about the special case of binary protocols, when $|\mathcal{P}| = 2$:

Lemma 4.19. *Every binary protocol is implementable.*

In the binary case, participant-based reachability is equivalent to standard reachability, because both participants are involved in every synchronous communication. Because protocols are deterministic, there exist no two distinct states in a binary protocol that are simultaneously reachable for either participant, and thus *CC* holds vacuously.

4.3 Synthesis

When proving soundness, we chose the canonical implementation as our witness to implementability. In other words, if a protocol satisfies *CC*, then the canonical implementation implements it. When proving completeness, we showed that *any* implementation would cause a violation to protocol fidelity or deadlock-freedom. In other words, if a protocol violates *CC*, then no implementation exists. Having established that *CC* precisely characterizes implementable protocols, we combine these observations to yield the following corollary:

Corollary 4.20 (Canonical implementation is all you need). *A protocol is implementable if and only if the canonical implementation implements it.*

For an implementable protocol, this fact serves as a criterion for synthesizing implementations: any implementation that is canonical will suffice. For the general class of protocols, synthesis is undecidable. However, for many expressive fragments of protocols that still feature infinite data, e.g. corresponding to symbolic finite automata [20, 77] and certain classes of timed and register automata [5, 13], one can simply use off-the-shelf determinization algorithms to compute canonical implementations [4, 84, 85].

5 Checking Implementability

Having established that *CC* is precise for protocol implementability, we next present sound and relatively complete algorithms to check *CC* for several protocol classes. We start with the most general case of symbolic protocols before considering decidable classes of finite-state protocols.

5.1 Symbolic Protocols

In the remainder of the section, we fix a symbolic protocol $\mathbb{S} = (S, R, \Delta, s_0, \rho_0, F)$. We assume that the concretization of $\mathbb{S}$ is a GCLTS (Definition 3.1). Additionally, we define two copies of the symbolic protocol, denoted $\mathbb{S}_1$ and $\mathbb{S}_2$ that we will use in describing our symbolic implementability check. Each copy $\mathbb{S}_i = (R_i, S, \Delta_i, \rho_i, s_0, F)$ with $i \in \{1, 2\}$ is obtained from $\mathbb{S}$ by renaming each register r to a fresh register r_i, each unique communication variable x to x_i, and substituting the new register and communication variables into the transition constraints and initial register assignment accordingly; the control states remain the same.

Because symbolic protocols describe concrete protocols with infinitely many states and transitions, implementability cannot be checked explicitly using our *CC* characterization for protocols, i.e. by iterating over all states and transitions. Instead, we present symbolic conditions that are valid on the symbolic protocol if and only if its concrete protocol is implementable.

Theorem 5.1 (Symbolic Implementability). *$\mathbb{S}$ is implementable if and only if it satisfies Symbolic Send Coherence, Symbolic Receive Coherence, and Symbolic No Mixed Choice.*

We now present these symbolic conditions, starting with Symbolic Send Coherence.

Send Coherence first requires us to characterize pairs of states in a protocol that are simultaneously reachable for each participant on some prefix in its local language. In the symbolic setting, this amounts to the following: given a participant and a pair of control states (s_1, s_2) in the symbolic protocol, characterize the register assignments for pairs of concrete states (s_1, ρ_1), (s_2, ρ_2) that are in the respective control states. The predicate $\mathsf{prodreach}_{\mathsf{p}}(s_1, \boldsymbol{r_1}, s_2, \boldsymbol{r_2})$ describes this for each $\mathsf{p} \in \mathcal{P}$ where $\boldsymbol{r_i}$ are vectors of the registers in R_i obtained by ordering them according to some fixed total order. We define this predicate as a least fixpoint as follows.

Definition 5.2 (Simultaneous reachability in product symbolic protocol). Let $\mathsf{p} \in \mathcal{P}$ be a participant and let $s_1, s_1', s_2, s_2' \in S$. Then,

$$\begin{aligned}\mathsf{prodreach}_{\mathsf{p}}(s_1', \boldsymbol{r_1'}, s_2', \boldsymbol{r_2'}) \coloneqq_{\mu}\ & (\ s_1' = s_0 \wedge s_2' = s_0 \wedge \boldsymbol{r_1'} = \rho_0 \wedge \boldsymbol{r_2'} = \rho_0\ ) \\ \vee\ & \Big(\bigvee_{\substack{(s_1,\, \mathsf{r}\to\mathsf{s}:x_1\{\varphi_1\},\, s_1')\in\Delta_1 \\ (s_2,\, \mathsf{r}\to\mathsf{s}:x_2\{\varphi_2\},\, s_2')\in\Delta_2 \\ \mathsf{p}=\mathsf{r}\vee\mathsf{p}=\mathsf{s}}} \exists x_1 x_2 \boldsymbol{r_1} \boldsymbol{r_2}.\, \mathsf{prodreach}_{\mathsf{p}}(s_1, \boldsymbol{r_1}, s_2, \boldsymbol{r_2}) \wedge \varphi_1 \wedge \varphi_2 \wedge x_1 = x_2\ \Big) \\ \vee\ & \Big(\bigvee_{(s_1,\, \mathsf{r}\to\mathsf{s}:x_1\{\varphi_1\},\, s_1')\in\Delta_1\,\wedge\,\mathsf{p}\neq\mathsf{r}\wedge\mathsf{p}\neq\mathsf{s}} \exists x_1 \boldsymbol{r_1}.\, \mathsf{prodreach}_{\mathsf{p}}(s_1, \boldsymbol{r_1}, s_2', \boldsymbol{r_2'}) \wedge \varphi_1\ \Big) \\ \vee\ & \Big(\bigvee_{(s_2,\, \mathsf{r}\to\mathsf{s}:x_2\{\varphi_2\},\, s_2')\in\Delta_2\,\wedge\,\mathsf{p}\neq\mathsf{r}\wedge\mathsf{p}\neq\mathsf{s}} \exists x_2 \boldsymbol{r_2}.\, \mathsf{prodreach}_{\mathsf{p}}(s_1', \boldsymbol{r_1'}, s_2, \boldsymbol{r_2}) \wedge \varphi_2\ \Big)\ .\end{aligned}$$

The second top-level disjunct in the definition after the base case handles the cases where $\mathbb{S}_1$ and $\mathbb{S}_2$ synchronize on a common action involving p. The remaining two disjuncts correspond to the cases where either $\mathbb{S}_1$ or $\mathbb{S}_2$ follows an ε transition.

Given a pair of simultaneously reachable states (s_1, ρ_1), (s_2, ρ_2) in p, Send Coherence now checks whether all values x_1 that can be sent to some q in (s_1, ρ_1) can also be sent from (s_2, ρ_2), modulo following ε transitions to reach the actual state where p can send to q. We thus need to express ε-reachability. We formalize the dual: the predicate $\mathsf{unreach}^{\varepsilon}_{\mathsf{p},\mathsf{q}}(s_2, \boldsymbol{r_2}, x_1)$ expresses that p *cannot* reach any state where it may send x_1 to q, by following ε transitions from symbolic state $(s_2, \boldsymbol{r_2})$. This is formulated as a greatest fixpoint as follows:

Definition 5.3 (ε-unreachability of p sending x to q). For $\mathsf{p}, \mathsf{q} \in \mathcal{P}$ and $s \in S$, let

$$\mathsf{unreach}^{\varepsilon}_{\mathsf{p},\mathsf{q}}(s, \boldsymbol{r}, x) \coloneqq_{\nu}\ \Big(\bigwedge_{(s,\, \mathsf{p}\to\mathsf{q}:y\{\varphi\},\, s')\in\Delta} \neg\varphi[x/y]\ \Big) \wedge \Big(\bigwedge_{\substack{(s,\, \mathsf{r}\to\mathsf{t}:y\{\varphi\},\, s')\in\Delta \\ \mathsf{p}\neq\mathsf{r}\wedge\mathsf{p}\neq\mathsf{t}}} \forall y\, \boldsymbol{r'}.\, \varphi \Rightarrow \mathsf{unreach}^{\varepsilon}_{\mathsf{p},\mathsf{q}}(s', \boldsymbol{r'}, x)\ \Big)\ .$$

The first conjunct checks that whenever p reaches a state with an outgoing send transition to q, it cannot send the value x because the transition constraint φ is not satisfied. The second conjunct checks that every outgoing ε transition is either disabled ($\neg\varphi$ holds) or following the transition does not reach an appropriate send state.

We combine the auxiliary predicates into our Symbolic Send Coherence condition.

Definition 5.4 (Symbolic Send Coherence). A symbolic protocol $\mathbb{S}$ satisfies Symbolic Send Coherence when for each transition $s_1 \xrightarrow{\mathsf{p}\to\mathsf{q}:x_1\{\varphi_1\}} s_1' \in \Delta_1$ and state $s_2 \in S$, the following is valid:

$$\mathsf{prodreach}_{\mathsf{p}}(s_1, \boldsymbol{r_1}, s_2, \boldsymbol{r_2})\ \wedge\ \varphi_1 \wedge \mathsf{unreach}^{\varepsilon}_{\mathsf{p},\mathsf{q}}(s_2, \boldsymbol{r_2}, x_1)\ \Longrightarrow\ \bot\ .$$

A keen reader may have noticed that because the symbolic characterization of Send Coherence involves a greatest fixpoint, it is a liveness property. Thus, proving Send Coherence, in general, involves a termination argument. To see this, consider the two protocols shown in Figs. 14 and 15. Consider the pair of states $(q_1, [c \mapsto 0])$ and $(q_3, [c \mapsto 0])$ which are simultaneously reachable for r in both protocols. The send transition for r enabled in q_1 needs to be matched with a corresponding send transition in an ε-reachable state from q_3. The only candidate states for this match in both protocols are those at control state q_4. These states are reachable from q_3 if and only if the loop in q_3 terminates, which it does in Fig. 14 but not in Fig. 15.

Receive Coherence is conditioned on two simultaneously reachable states $(s_1, \boldsymbol{r_1})$ and $(s_2, \boldsymbol{r_2})$ for a participant q. It checks that if q can receive x from p in the first state, q cannot also receive x as the first message from p in the second state, in which it can also receive from a different participant r, unless p sending x causally depends on q first receiving from r. We thus need to define a predicate that captures whether x_1 may be available as the first message from q to p, while tracking causal

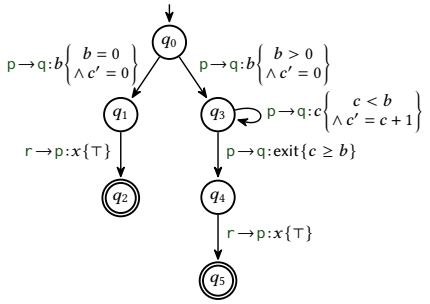

Fig. 14. Example where states q_1 and q_3 satisfy Send Coherence for r.

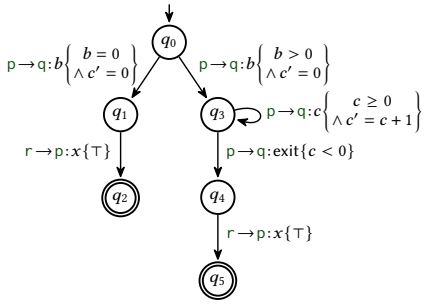

Fig. 15. Example where states q_1 and q_3 violate Send Coherence for r.

dependencies. We introduce a family of predicates $\mathsf{avail}_{\mathsf{p},\mathsf{q},\mathcal{B}}(x_1, s_2, \boldsymbol{r_2})$ for this purpose. Here, $\mathcal{B}$ is used to track the causal dependencies. $\mathcal{B}$ tracks the set of participants that are blocked from sending a message because their send action causally depends on q first receiving from r. The predicates are defined as the least fixpoint of the following mutually recursive definition.

Definition 5.5 (Symbolic Availability).

$$\begin{aligned}\mathsf{avail}_{\mathsf{p},\mathsf{q},\mathcal{B}}(x_1, s, \boldsymbol{r}) :=_\mu \quad & \Big(\bigvee_{\substack{(s,\, \mathsf{r}\to\mathsf{t}:x\{\varphi\},\, s') \in \Delta \\ \mathsf{r}\in\mathcal{B} \\ \mathsf{r}\neq\mathsf{p}\vee\mathsf{t}\neq\mathsf{q}}} \exists x\, \boldsymbol{r'}.\, \mathsf{avail}_{\mathsf{p},\mathsf{q},\mathcal{B}\cup\{\mathsf{t}\}}(x_1, s', \boldsymbol{r'}) \wedge \varphi \Big) \\ \vee \Big(& \bigvee_{\substack{(s,\, \mathsf{r}\to\mathsf{t}:x\{\varphi\},\, s') \in \Delta \\ \mathsf{r}\notin\mathcal{B} \\ \mathsf{r}\neq\mathsf{p}\vee\mathsf{t}\neq\mathsf{q}}} \exists x\, \boldsymbol{r'}.\, \mathsf{avail}_{\mathsf{p},\mathsf{q},\mathcal{B}}(x_1, s', \boldsymbol{r'}) \wedge \varphi \Big) \vee \Big(\bigvee_{\substack{(s,\, \mathsf{p}\to\mathsf{q}:x\{\varphi\},\, s') \in \Delta \\ \mathsf{p}\notin\mathcal{B}}} \varphi[x_1/x] \Big)\ .\end{aligned}$$

The last disjunct in the definition handles the cases where the message x_1 from p is immediately available to be received by q in symbolic state $(s, \boldsymbol{r})$ and p has not been blocked from sending. The other two disjuncts handle the cases when x_1 becomes available after some other message exchange between r and t. Here, if r is blocked, then t also becomes blocked since it depends on r sending before it can receive (the first disjunct). Otherwise, no participant is added to the blocked set (the second disjunct).

With the available message predicate in place, we can now define Symbolic Receive Coherence.

Definition 5.6 (Symbolic Receive Coherence). A symbolic protocol $\mathbb{S}$ satisfies Symbolic Receive Coherence when for every pair of transitions $s_1 \xrightarrow{\mathsf{p}\to\mathsf{q}:x_1\{\varphi_1\}} s_1' \in \Delta_1$ and $s_2 \xrightarrow{\mathsf{r}\to\mathsf{q}:x_2\{\varphi_2\}} s_2' \in \Delta_2$ with $\mathsf{p} \neq \mathsf{r}$:

$$\mathsf{prodreach}_{\mathsf{q}}(s_1, \boldsymbol{r_1}, s_2, \boldsymbol{r_2}) \ \wedge\ \varphi_1 \ \wedge\ \varphi_2 \ \wedge\ \mathsf{avail}_{\mathsf{p},\mathsf{q},\{\mathsf{q}\}}(x_1, s_2', \boldsymbol{r_2'}) \implies \bot\ .$$

Finally, No Mixed Choice is conditioned on two simultaneously reachable states $(s_1, \boldsymbol{r_1})$ and $(s_2, \boldsymbol{r_2})$ with outgoing send and receive transitions for a participant p.

Definition 5.7 (Symbolic No Mixed Choice). A symbolic protocol $\mathbb{S}$ satisfies Symbolic No Mixed Choice when for every pair of transitions $s_1 \xrightarrow{\mathsf{p}\to\mathsf{q}:x_1\{\varphi_1\}} s_1' \in \Delta_1$ and $s_2 \xrightarrow{\mathsf{r}\to\mathsf{p}:x_2\{\varphi_2\}} s_2' \in \Delta_2$:

$$\mathsf{prodreach}_{\mathsf{p}}(s_1, \boldsymbol{r_1}, s_2, \boldsymbol{r_2}) \ \wedge\ \varphi_1 \ \wedge\ \varphi_2 \implies \bot\ .$$

We conclude this section with a discussion of how to check GCLTS assumptions, namely sink finality, sender-driven choice, and deadlock-freedom, on a symbolic protocol. Sink finality can be

Algorithm 1 Check *CC* for finite protocols

▷ *Let LTS* $\mathcal{S} = (S, \Gamma_{sync}, T, s_0, F)$
▷ *Checking Send Coherence*
for $s_1 \xrightarrow{\mathsf{p}\to\mathsf{q}:m} s_2 \in T$ **do**
  for $s \neq s_1 \in S$ **do**
    if $\mathcal{L}(S, \Gamma_\mathsf{p} \uplus \{\varepsilon\}, T_\mathsf{p}, s_0, \{s\}) \cap \mathcal{L}(S, \Gamma_\mathsf{p} \uplus \{\varepsilon\}, T_\mathsf{p}, s_0, \{s_1\}) \neq \emptyset$ **then**
      $b \leftarrow \bot$
      for $s_3 \xrightarrow{\mathsf{p}\to\mathsf{q}:m} s_4 \in T$ **do** $b \leftarrow b \vee \left(s \underset{\mathsf{p}}{\overset{\varepsilon}{\Longrightarrow}}{}^{*} s_3\right)$
      if $\neg b$ **then return** $\bot$
▷ *Checking Receive Coherence*
for $s_1 \xrightarrow{\mathsf{p}\to\mathsf{q}:m} s_2, s_3 \xrightarrow{\mathsf{r}\to\mathsf{q}:m} s_4 \in T, s_1 \neq s_2, \mathsf{p} \neq \mathsf{r}$ **do**
  if $\mathcal{L}(S, \Gamma_\mathsf{q} \uplus \{\varepsilon\}, T_\mathsf{q}, s_0, \{s_1\}) \cap \mathcal{L}(S, \Gamma_\mathsf{q} \uplus \{\varepsilon\}, T_\mathsf{q}, s_0, \{s_3\}) \neq \emptyset$ **then**
    if $\mathsf{avail}_{\mathsf{p},\mathsf{q},\{\mathsf{q}\}}(m, s_4)$ **then return** $\bot$
▷ *Checking No Mixed Choice*
for $s_1 \xrightarrow{\mathsf{p}\to\mathsf{q}:m} s_2, s_3 \xrightarrow{\mathsf{r}\to\mathsf{p}:m} s_4 \in T, s_1 \neq s_2$ **do**
  if $\mathcal{L}(S, \Gamma_\mathsf{q} \uplus \{\varepsilon\}, T_\mathsf{q}, s_0, \{s_1\}) \cap \mathcal{L}(S, \Gamma_\mathsf{q} \uplus \{\varepsilon\}, T_\mathsf{q}, s_0, \{s_3\}) \neq \emptyset$ **then return** $\bot$
return $\top$

checked directly by examining the syntax of the symbolic protocol. Sender-driven choice without determinism can likewise be checked directly on the states of the symbolic protocol. Determinism and deadlock freedom are undecidable in general but can both be reduced to reachability. Thus, both our Symbolic Coherence Conditions and GCLTS assumptions can be discharged using off-the-shelf μCLP solvers. We leave such an implementation to future work.

We next apply our framework to decidable fragments of symbolic protocols, some of which have been studied in the literature.

5.2 Finite Protocols

We first consider finite protocols. Let $\mathcal{S} = (S, \Gamma_{sync}, T, s_0, F)$ be a protocol with finite S and T. Because S and T are finite, we can transform *CC* into an imperative algorithm (see Algorithm 1) and use it to check implementability directly. For checking Receive Coherence, we need to decide the predicate $\mathsf{avail}_{\mathsf{p},\mathsf{q},\{\mathsf{q}\}}(m, s)$, which is defined like the symbolic availability predicate $\mathsf{avail}_{\mathsf{p},\mathsf{q},\{\mathsf{q}\}}(x, s, \boldsymbol{r})$, except on protocols instead of symbolic protocols.

It is easy to see that Send Coherence and No Mixed Choice can be checked in time polynomial in the size of $\mathcal{S}$. However, the inclusion of $\mathsf{avail}_{\mathsf{p},\mathsf{q},\{\mathsf{q}\}}(m, s)$ as a subroutine for checking Receive Coherence yields the following complexity result.

Theorem 5.8. *Implementability of finite protocols is co-NP-complete.*

Proof. To see that implementability is in co-NP, observe that violations of Send Coherence and No Mixed Choice can be checked in NP, by guessing a participant p and a pair of states s_1, s_2 that satisfy the respective preconditions, and verifying simultaneous reachability of s_1 and s_2 for p. For Send Coherence, we guess an additional state s_3 with an outgoing transition labeled with $\mathsf{p} \to \mathsf{q} : m$, and check ε-reachability from s_1 to s_3. For Receive Coherence, $\mathsf{avail}_{\mathsf{p},\mathsf{q},\{\mathsf{q}\}}(m, s_2)$ can be checked in NP by guessing a simple path in $\mathcal{S}$ from s_2 to some state s' with an outgoing transition labeled with $\mathsf{p} \to \mathsf{q} : m$. We then evaluate $\mathsf{avail}_{\mathsf{p},\mathsf{q},\{\mathsf{q}\}}(m, s_2)$ along that path, which can be done in polynomial time. We can restrict ourselves to simple paths because the blocked set $\mathcal{B}$ monotonically increases when traversing a path in $\mathcal{S}$. Moreover, $\mathsf{avail}_{\mathsf{p},\mathsf{q},\{\mathsf{q}\}}(m, s_2)$ is antitone in the blocked set.

We show NP-hardness of non-implementability via a reduction from the 3-SAT problem. Assume a 3-SAT instance $\varphi = C_1 \wedge \ldots \wedge C_k$. Let $x_1, \ldots, x_n$ be the variables occurring in φ and let L_{ij} be the

jth literal of clause C_i, with $1 \le i \le k$ and $1 \le j \le 3$. We construct a protocol $\mathcal{S}_\varphi$ over participants $\mathcal{P} = \{\mathtt{p}, \mathtt{q}, \mathtt{r}, \mathtt{x}_1, \overline{\mathtt{x}}_1, \ldots, \mathtt{x}_n, \overline{\mathtt{x}}_n\}$, such that φ is satisfiable iff $\mathcal{S}_\varphi$ is implementable. In particular, we ensure that $\mathcal{S}_\varphi$ is implementable iff $\mathsf{avail}_{\mathtt{p},\mathtt{q},\{\mathtt{q}\}}(m, s)$ does not hold for some state s in $\mathcal{S}_\varphi$. The protocol $\mathcal{S}_\varphi$ is constructed from the following subprotocols:

(1) Define a protocol $\mathcal{S}_X$ representing a truth assignment to variables x_i with states $s_1, \ldots, s_{n+1}$ as follows: for every $1 \le i \le n$ there are two paths of four transitions each between s_i and s_{i+1}. The paths consist of transitions labeled with $\mathtt{r} \to \mathtt{x}_\mathtt{i} : \bot$, $\mathtt{r} \to \overline{\mathtt{x}}_\mathtt{i} : \top$, $\mathtt{r} \to \mathtt{q} : m_{x_i}$, $\mathtt{q} \to \mathtt{x}_\mathtt{i} : m$, and $\mathtt{r} \to \overline{\mathtt{x}}_\mathtt{i} : \bot$, $\mathtt{r} \to \mathtt{x}_\mathtt{i} : \top$, $\mathtt{r} \to \mathtt{q} : m_{\overline{x}_i}$, $\mathtt{q} \to \overline{\mathtt{x}}_\mathtt{i} : m$, respectively.
(2) Define a protocol $\mathcal{S}_C$ representing the clauses C_i with states $t_1, \ldots, t_{k+1}$ as follows. For each $1 \le i \le k$ there are three paths of three transitions between each t_i and t_{i+1}, one for each $1 \le j \le 3$, labeled with $\mathtt{r} \to s : m_j$, $\mathtt{r} \to \mathtt{p} : m_r$, $s \to \mathtt{p} : m$, where $s = \mathtt{x}$ if $L_{ij} = x$ and $s = \overline{\mathtt{x}}$ if $L_{ij} = \neg x$ for $x \in \{x_1, \ldots, x_n\}$.
(3) Define a protocol $\mathcal{S}_F$ with two states q'_f and q_f and a single transition from q'_f to q_f labeled with $\mathtt{p} \to \mathtt{q} : m$.
(4) Define a protocol $\mathcal{S}_T$ with five states $q_1, \ldots, q_5$, and two paths from q_1, respectively $q_1 \xrightarrow{\mathtt{r}\to\mathtt{p}:m_1} q_2 \xrightarrow{\mathtt{r}\to\mathtt{q}:m} q_3$ and $q_1 \xrightarrow{\mathtt{r}\to\mathtt{p}:m_2} q_4 \xrightarrow{\mathtt{p}\to\mathtt{q}:m} q_5$.

We merge all of the above protocols to obtain $\mathcal{S}_\varphi$ by identifying the state q_3 with s_1, s_{n+1} with t_1 and t_{k+1} with q'_f. The initial state is q_1 and the final states are $\{q_5, q_f\}$.

Observe that the size of $\mathcal{S}_\varphi$ is linear in the size of φ. Moreover, it is easy to check that $\mathcal{S}_\varphi$ is indeed a GCLTS: all choices are sender-driven and deterministic, and final states are the only states with no outgoing transitions, yielding sink-finality and deadlock-freedom.

We first establish that $\mathsf{avail}_{\mathtt{p},\mathtt{q},\{\mathtt{q}\}}(m, q_3)$ holds in $\mathcal{S}_\varphi$ iff φ is satisfiable. Observe that the blocked set $\mathcal{B}$ computed by $\mathsf{avail}_{\mathtt{p},\mathtt{q},\{\mathtt{q}\}}(m, q_3)$ along a path between s_1 and s_{n+1} contains for each variable x_i either $\mathtt{x}_\mathtt{i}$ or $\overline{\mathtt{x}}_\mathtt{i}$. The blocked set $\mathcal{B}$ thus encodes a truth assignment $\rho_\mathcal{B}$ for the x_i's where $\rho_\mathcal{B}(x_i) = \top$ iff $\mathtt{x}_\mathtt{i} \notin \mathcal{B}$. By construction of $\mathcal{S}_X$, for every truth assignment ρ, there exists a path between s_1 and s_{n+1} such that $\rho = \rho_\mathcal{B}$ for the blocked set $\mathcal{B}$ computed along that path.

The paths between states t_i and t_{i+1} in subprotocol $\mathcal{S}_C$ allow $\mathtt{p}$ to proceed and not be blocked if one of the paths has a participant not in $\mathcal{B}$, i.e. C_i is satisfied by $\rho_\mathcal{B}$. Thus, a path from $s_{n+1} = t_1$ to $t_{k+1} = q'_f$ adds $\mathtt{p}$ to $\mathcal{B}$ at t_i iff $\rho_\mathcal{B}$ does not satisfy at least one of the clauses C_i. Therefore, m is available in q_3 iff there exists a $\mathcal{B}$ such that $\rho_\mathcal{B}$ satisfies φ.

It remains to show that $\mathcal{S}_\varphi$ is non-implementable iff $\mathsf{avail}_{\mathtt{p},\mathtt{q},\{\mathtt{q}\}}(m, q_3)$ holds in $\mathcal{S}_\varphi$. We argue that all participants except $\mathtt{q}$ have sufficient local information about the control flow of the protocol to behave accordingly. Participant $\mathtt{r}$ dictates the control flow at every branching point of the protocol, and thus is implementable. Participants $\mathtt{x}_1, \overline{\mathtt{x}}_1, \ldots \mathtt{x}_n, \overline{\mathtt{x}}_n$ learn the control flow via receiving messages from participant $\mathtt{r}$, whose labels uniquely determine their next actions: receiving $\top$ means inaction, receiving $\bot$ means receive a further message from $\mathtt{q}$, and receiving m means send a message encoding its own variable name to $\mathtt{p}$. Participant $\mathtt{p}$ is likewise informed by $\mathtt{r}$ about the control flow, and only sends m to $\mathtt{q}$ upon either receiving m_2 or *top* from $\mathtt{r}$. Upon receiving $\mathtt{r}$'s choice of disjunct for each clause, it anticipates a message from the participant encoding that disjunct.

Participant $\mathtt{q}$, on the other hand, is not informed by $\mathtt{r}$ about $\mathtt{r}$'s initial choice at G_{x_1}, and can locally choose between receptions from $\mathtt{p}$ or $\mathtt{r}$. In the case that $\mathsf{avail}_{\mathtt{p},\mathtt{q},\{\mathtt{q}\}}(m, q_3)$ holds, there exists a path from $\overline{G}$ to $\mathbf{G}_\varphi$ in which $\mathtt{p}$ is not blocked. Thus, the message from $\mathtt{p}$ can be asynchronously reordered to arrive in $\mathtt{q}$'s channel such that both receptions are enabled, and $\mathtt{q}$ may violate implementability by receiving the message from $\mathtt{p}$ out of order. If $\mathsf{avail}_{\mathtt{p},\mathtt{q},\{\mathtt{q}\}}(m, q_3)$ does not hold, only one reception is enabled, which uniquely informs $\mathtt{q}$ about $\mathtt{r}$'s choice. In the case that the reception from $\mathtt{p}$ is enabled, $\mathtt{q}$ terminates, otherwise it receives messages from $\mathtt{r}$ encoding participants to send further messages

to, and terminates upon receiving the final message from p. Thus, $\mathcal{S}_\varphi$ is non-implementable iff q violates Receive Coherence for the transitions $q_2 \xrightarrow{\mathsf{r}\to\mathsf{q}:m} q_3$ and $q_4 \xrightarrow{\mathsf{p}\to\mathsf{q}:m} q_5$, i.e. $\mathsf{avail}_{\mathsf{p},\mathsf{q},\{\mathsf{q}\}}(m, q_3)$ does not hold.

We obtain that $\mathcal{S}_\varphi$ is non-implementable iff $\mathsf{avail}_{\mathsf{p},\mathsf{q},\{\mathsf{q}\}}(m, q_3)$ holds in $\mathcal{S}_\varphi$ iff φ is satisfiable. □

Implementability for global multiparty session types was shown in [56] to be in PSPACE, with the matching lower bound corrected in [57]. We show that, in fact, the same 3-SAT reduction can be adapted to show co-NP-completeness of implementability for global multiparty session types.

Global Multiparty Session Types. Global types for MSTs [56] are defined by the grammar:

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

where p, q_i range over $\mathcal{P}$, m_i over a finite set $\mathcal{V}$, and t over a set of recursion variables. The semantics of a global type $\mathbf{G}$ are defined using a finite state machine $\mathsf{GAut}(\mathbf{G}) = (Q_\mathbf{G}, \Gamma_{sync} \uplus \{\varepsilon\}, \delta_\mathbf{G}, q_{0,\mathbf{G}}, F_\mathbf{G})$ 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 $(\sum_{i\in I} \mathsf{p}\to\mathsf{q}_i : m_i.G_i, \mathsf{p}\to\mathsf{q}_i : m_i, G_i)$ for each $i \in I$, as well as $(\mu t.G', \varepsilon, G')$ and $(t, \varepsilon, \mu t.G')$ for each subterm $\mu t.G'$, $q_{0,\mathbf{G}} = \mathbf{G}$ and $F_\mathbf{G} = \{0\}$.

Each branch of a choice is assumed to be distinct: $\forall i, j \in I.\, i \neq j \implies (\mathsf{q}_i, m_i) \neq (\mathsf{q}_j, m_j)$, and the sender and receiver of an atomic action is assumed to be distinct: $\forall i \in I.\, \mathsf{p} \neq \mathsf{q}_i$. Recursion is guarded: in $\mu t.\, G$, there is at least one message between μt and each t in G.

Each ε transition in $\mathsf{GAut}(\mathbf{G})$ is the only transition from the state it originates from. This makes removing them easy, yielding a protocol $\mathcal{S}_\mathbf{G} = (Q_\mathbf{G}, \Gamma_{sync}, \delta'_\mathbf{G}, q_{0,\mathbf{G}}, F_\mathbf{G})$, where $\delta'_\mathbf{G}$ contains only transitions labeled with $l \in \Gamma_{sync}$. It is easy to verify that $\mathcal{S}_\mathbf{G}$ is indeed a GCLTS.

Lemma 5.9. *Implementability of global types is co-NP-complete.*

Our reduction shows that deciding the $\mathsf{avail}_{\mathsf{p},\mathsf{q},\{\mathsf{q}\}}(m, s)$ predicate for global types is in co-NP, which contradicts the polynomial time upper bound claimed in [55]. The proof of Lemma 5.9 can be found in Appendix B of the extended version [58].

5.3 Symbolic Finite Protocols

Finally, we study symbolic representations of finite protocols. More precisely, we consider the fragment of symbolic protocols where $\mathcal{V}$ is the set of Booleans and all transition constraints φ are given by propositional formulas. We show that for this class of symbolic protocols, the implementability problem is PSPACE-complete.

Theorem 5.10. *Implementability of symbolic finite protocols is PSPACE-complete.*

Proof sketch. To show that implementability is in PSPACE, we show that a witness to the negation of *CC* can be checked in nondeterministic polynomial space. This follows by a reduction to the reachability problem for extended finite state machines, which is in PSPACE [36]. By Savitch's Theorem, it follows that the negation of *CC* is in PSPACE. Because PSPACE is closed under complement and *CC* precisely characterizes implementability, it follows that implementability is in PSPACE.

We show PSPACE-hardness of the implementability problem by a reduction from the PSPACE-hard problem of deciding reachability for 1-safe Petri nets [27]. Let (N, M_0) be a 1-safe Petri net, with $N = (S, T, F)$. Let M be a marking of N.

We construct a symbolic protocol that is implementable iff N does not reach M. For ease of exposition, we present this symbolic protocol as a symbolic dependent global type $\mathbf{G}_N$ with the understanding that the encoding of $\mathbf{G}_N$ as a symbolic protocol is clear.

We first describe the construction of $\mathbf{G}_N$. The outermost structure of $\mathbf{G}_N$ consists of a participant $\mathtt{r}$ communicating a choice between two branches to $\mathtt{s}$ where the bottom branch solely consists of $\mathtt{p}$ sending l to $\mathtt{q}$: $\mathbf{G}_N \coloneqq (\mathtt{r} \to \mathtt{s} : m_1\{\top\}.\, G_t + \mathtt{r} \to \mathtt{s} : m_2\{\top\}.\, \mathtt{p} \to \mathtt{q} : l\{\top\}.\, 0)$. Since $\mathtt{p}$ is not informed about the choice of the branch taken by $\mathtt{s}$, it will have to be able to match this send transition in every run that follows the continuation G_t of the top branch. We will construct G_t such that this match is possible iff M is reachable in N.

In G_t, participants $\mathtt{r}$ and $\mathtt{s}$ enter a loop that simulates N:

$$G_t \coloneqq \mu s[v \coloneqq M_0].\ + \begin{cases} \sum_{t \in T}\ \mathtt{r} \to \mathtt{s} : m_t\{v \Rightarrow t^-\}.\, s[v \coloneqq ((v \wedge \neg t^-) \vee t^+)] \\ \mathtt{r} \to \mathtt{s} : \mathit{restart}\{\top\}.\, s[v \coloneqq M_0] \\ \mathtt{r} \to \mathtt{s} : \mathit{reach}_M\{v = M\}.\, \mathtt{p} \to \mathtt{q} : l\{\top\}.\, 0 \end{cases}$$

The loop variable v is a $|S|$-length bitvector that tracks the current marking of the net. It is initialized to M_0. Inside the loop, $\mathtt{r}$ has the following choices. First, it may pick any transition $t \in T$ of the net and send an m_t message to $\mathtt{s}$, provided the transition is enabled for firing (i.e., the input places of t all contain a token: $v \Rightarrow t^-$). After this communication, v is updated according to the fired transition t.

The last branch of the choice in the loop is enabled if v is equal to M. Here, $\mathtt{r}$ can send reach_M to $\mathtt{s}$, which gives $\mathtt{p}$ the opportunity to send the l message to $\mathtt{q}$, allowing it to match the send transition from the lower branch in the top level choice of G_N.

Finally, the middle branch allows $\mathtt{r}$ to abort the simulation at any point and start over. This ensures that if the simulation ever reaches a dead state due to firing a transition that would render M unreachable, it can recover by starting again from M_0. Thus, for all states of the simulator, $\mathtt{p}$ has an ε path from that state to a state where it can send l to $\mathtt{q}$ iff M is reachable from M_0 in N. The only other sender is $\mathtt{r}$ which makes all choices and, hence, never reaches two different states along the same prefix trace, thus satisfying Send Coherence trivially. It follows that Send Coherence for $\mathtt{p}$ holds iff M is reachable from M_0 in N. To see that Receive Coherence holds, observe that no participant receives messages from two different senders. No Mixed Choice similarly holds trivially.

G_N is deadlock-free because the branch in the loop of G_t where $\mathtt{r}$ sends the *restart* message is always enabled. Moreover, it is easy to see that G_N is deterministic because each branch of a choice sends a different message value.

In summary, G_N is a GCLTS that is implementable iff N reaches M. The size of $\mathbf{G}_N$ is linear in the size of N, so we obtain the desired reduction. □

6 Related Work

Table 1 summarizes the most closely related works that address the implementability problem of communication protocols with data refinements. We discuss these works in terms of key expressive features and completeness of characterization.

Table 1. Comparison of related work (in chronological order)

Paper	Communication paradigm	Branching restrictions	History sensitivity	Characterization
[7]	asynchronous	directed choice	required	incomplete
[6]	asynchronous	directed choice	required	incomplete
[81]	synchronous	directed choice	required	incomplete
[89]	synchronous	directed choice	required	incomplete
[34]	synchronous	well-sequencedness	required	unknown
this work	asynchronous	sender-driven choice	not required	relatively complete

Expressivity. All existing works in Table 1 effectively require *history-sensitivity*, which means that a "predicate guaranteed by a [participant p] can only contain those interaction variables that [p] knows" [7], see also [6, Def. 2]. As discussed in §4, syntactic approaches to analyzing variable knowledge is overly conservative, and as a result no prior work can handle protocols such as the example in Fig. 12. In a similar vein, Zhou et al. [89] impose the syntactic restriction that all participants in a loop must be able to update all loop registers, which rules out loops like the one in the two-bidder protocol (Fig. 1).

Furthermore, all prior works except for [34] employ the directed choice restriction, which is strictly less general than sender-driven choice. Many of these works also feature separate constructs for selecting branches and sending data. In our symbolic protocols, this is not necessary because selecting branches can be modeled with equality predicates, as demonstrated by Fig. 7. Gheri et al. [34] generalize choreography automata, which are finite-state LTSs with communciation events as transition labels but without final states. One major difference between our work and theirs lies in the treatment of interleavings. Unlike our protocol semantics, which are closed under the indistinguishability relation $\sim$, inspired by Lamport's happened-before relation, choreography automata languages do not include any interleavings not present in the language. Setting aside asynchronous traces, the protocol $\mathtt{p} \to \mathtt{q} : m.\, \mathtt{r} \to \mathtt{s} : m.\, 0$ in our setting would need to be represented as $\mathtt{p} \to \mathtt{q} : m.\, \mathtt{r} \to \mathtt{s} : m.\, 0 \;+\; \mathtt{r} \to \mathtt{s} : m.\, \mathtt{p} \to \mathtt{q} : m.\, 0$ in their setting, and the following protocol $\mu t.\, \mathtt{p} \to \mathtt{q} : m.\, \mathtt{r} \to \mathtt{s} : m.\, t$ does not admit a representation as a choreography automaton. The branching behaviors are restricted with a well-sequencedness condition [34, Def. 3.2], a condition that has since been refined because it was shown to be flawed [29]. Majumdar et al. [61] showed that well-formedness conditions on synchronous choreography automata do not generalize soundly to the asynchronous setting.

Asynchronous communication is more challenging to analyze in general because it easily gives rise to infinite-state systems. Zhou [88] conjectures that the framework in [89] "can be extended to support asynchronous communication", but does not conjecture if and how the projection operator would change. Due to directed choice, the same projection operator may remain sound under asynchronous semantics, because it rules out protocols where participants have a choice to receive from different senders. However, it will also likely inherit the same sources of incompleteness present in the synchronous setting.

In contrast to all aforementioned works, several works [9, 10, 17] allow to specify send and receive events separately with "deconfined" global types. Deconfined global types are specified as a parallel composition of local processes, and then checked for desirable correctness properties, which were shown to be undecidable [17].

Completeness. Implementability is a thoroughly-studied problem in the high-level message sequence chart (HMSC) literature. HMSCs are a standardized formalism for describing communication protocols in industry [82] and are well-studied in academia [30–32, 64, 74]. In the HMSC setting, implementability is called safe realizability, and is defined with respect to the implementation model of communicating finite state machines [8]. Similar to our setting, a canonical implementation exists for any HMSC [1, Thm. 13]; unlike our setting, it is always computable. Therefore, existing work has focused less on synthesis and more on checking implementability. Despite having only finite states and data, HMSC implementability was shown to be undecidable in general [59]. Various fragments have since been identified in which the problem regains decidability. Lohrey [59] showed implementability to be EXPSPACE-complete for bounded HMSCs [3, 68] and globally-cooperative HMSCs [33, 66]. These fragments restrict the communication topology of loops to be strongly and weakly connected respectively. For HMSCs where every two consecutive communications share a participant, implementability was shown to be PSPACE-complete [59].

In contrast, works that study comparably expressive protocol fragments to ours often sidestep the implementability question. Instead, implementability is addressed in the form of syntactic well-formedness conditions, as mentioned above, or indirectly through synthesis. None of the prior works attempted to show completeness; it was later shown in [56, 78] that all but Gheri et al. [34] are incomplete. Several works [6, 7, 81, 89] synthesize local implementations using the "classical" projection from multiparty session types. One kind of merge operator, called the plain merge, allows only the two participants in a choice to exhibit different behavior on each branch, a condition which is breached by our two-bidder protocol (Fig. 1). Zhou et al. [89] proves the soundness of projection with plain merge, but implements a more permissive variant called full merge in the toolchain. However, the projected local types are not guaranteed to be implementable: both Fig. 11 and Fig. 12 are projectable in [89]. Thus, the implementability problem is deferred to local types.

Our results show that synthesis is "as possible as" the determinization of the non-deterministic underlying automata fragment. This means that implementations can be synthesized even for expressive classes of protocols that correspond to e.g. symbolic finite automata [20, 77] and certain classes of timed and register automata [5, 13] due to the existence of off-the-shelf determinization algorithms for these classes [4, 84, 85].

Scalas and Yoshida [76] check safety properties of collections of local types by encoding the properties as μ-calculus formulas and then model checking the typing context against the specification. They focus primarily on finite-state typing contexts under synchronous semantics, and thus all properties in their setting are decidable. For the asynchronous setting, only three sound approximations of safety are proposed, one of which bounds channel sizes and thus falls back into the finite-state setting.

Next, we discuss further related works on choreographic programming and binary session types.

Choreographic Programming. Choreographic programming [15, 35, 43] describes global message-passing behaviors as programs rather than protocols, and therefore incorporate many more programming language features that are abstracted away in our model, such as computation and mutable state, in addition to features that our model cannot express, such as higher-order computations and delegation. Endpoint projection for choreographic programs, which shares a theoretical basis with multiparty session type projection, then generates individual, executable programs for each participant. The question of implementability, though undecidable in the presence of such expressivity, remains relevant to the soundness of endpoint projections. We discuss three approaches to endpoint projection. Pirouette [42] requires the programmer to specify explicit synchronization messages to ensure that "different locations stay in lock-step with each other", and conservatively rejects programs that are underspecified in this regard. Pirouette provides a mechanized proof of deadlock freedom for endpoint projections in Coq. Note that the claims of soundness and completeness in [42] are not with respect to implementability, but with respect to the translation via endpoint projection. HasChor [77] rules out non-implementability by automatically incorporating location broadcasts when a choice is made. No formal correctness claims are made in [77]. Jongmans and van den Bos [50] allow if- and while- statements to be annotated with a conjunction of conditional choices for each participant, which expresses decentralized decision-making in protocols. They show that their endpoint projection for well-formed choreographies guarantees deadlock freedom and functional correctness. All aforementioned choreographic programming works assume a synchronous network.

Binary Session Types with Refinements. Finally, we briefly mention work on binary session types with refinements and data dependencies. In the binary setting, implementability is a less interesting problem due to the inherent duality between the two protocol participants; the distinction between

global and local types is no longer meaningful. Griffith and Gunter [38] refine binary sessions with basic data types, and shows decidability of the subtyping problem. Gommerstadt et al. [37] applies a similar type system for runtime monitoring of binary communication. Thiemann and Vasconcelos [80] propose a label-dependent binary session type framework which allows the subsequent behavior of the protocol to depend on previous labels, which are drawn from a finite set. Das and Pfenning [22] study the undecidable problem of local type equality, and provide a sound approximate algorithm. Das et al. [21, 23] further apply binary session types with refinements to resource analysis of blockchain smart contracts and amortized cost analysis.

Actris [39] embeds binary session types into the Iris framework [51]. The framework assumes asynchronous communication with FIFO channels, and can verify programs that combine message-passing concurrency and shared-memory concurrency. Actris has been extended with session type subtyping (Actris 2.0 [40]) and with linearity to prove both preservation and progress (LinearActris [49]). Multris [41] is an extension of Actris in Iris to the multiparty setting. The message-passing layer of Multris is more restricted than Actris: Multris assumes synchronous communication and prohibits choice over channels: choices can only be made about message values between a given sender and receiver. Multris takes a bottom-up approach [76] to correctness: given a collection of local types, the type system checks that they can be safely combined. Multris guarantees protocol fidelity but not progress.

Data-Availability Statement

The extended version of this paper containing complete proofs can be found at [58].

Acknowledgments

This work is supported in parts by the National Science Foundation under the grant agreement 2304758 and by the Luxembourg National Research Fund (FNR) under the grant agreement C22/IS/17238244/AVVA. We thank the anonymous OOPSLA 2025 reviewers for their comments which improved the paper, and for identifying an erroneous claim in an earlier draft related to the complexity analysis of MST implementability.

References

[1] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. 2003. Inference of Message Sequence Charts. *IEEE Trans. Software Eng.* 29, 7 (2003), 623–633. https://doi.org/10.1109/TSE.2003.1214326

[2] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. 2005. Realizability and verification of MSC graphs. *Theor. Comput. Sci.* 331, 1 (2005), 97–114. https://doi.org/10.1016/J.TCS.2004.09.034

[3] Rajeev Alur and Mihalis Yannakakis. 1999. Model Checking of Message Sequence Charts. In *CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24-27, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1664)*, Jos C. M. Baeten and Sjouke Mauw (Eds.). Springer, 114–129. https://doi.org/10.1007/3-540-48320-9_10

[4] Nathalie Bertrand, Patricia Bouyer, Thomas Brihaye, and Pierre Carlier. 2018. When are stochastic transition systems tameable? *J. Log. Algebraic Methods Program.* 99 (2018), 41–96. https://doi.org/10.1016/J.JLAMP.2018.03.004

[5] Nathalie Bertrand, Amélie Stainer, Thierry Jéron, and Moez Krichen. 2015. A game approach to determinize timed automata. *Formal Methods Syst. Des.* 46, 1 (2015), 42–80. https://doi.org/10.1007/S10703-014-0220-1

[6] Laura Bocchi, Romain Demangeon, and Nobuko Yoshida. 2012. A Multiparty Multi-session Logic. In *Trustworthy Global Computing - 7th International Symposium, TGC 2012, Newcastle upon Tyne, UK, September 7-8, 2012, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8191)*, Catuscia Palamidessi and Mark Dermot Ryan (Eds.). Springer, 97–111. https://doi.org/10.1007/978-3-642-41157-1_7

[7] Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. 2010. A Theory of Design-by-Contract for Distributed Multiparty Interactions. In *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6269)*, Paul Gastin and François Laroussinie (Eds.). Springer, 162–176. https://doi.org/10.1007/978-3-642-15375-4_12

[8] Daniel Brand and Pitro Zafiropulo. 1983. On Communicating Finite-State Machines. *J. ACM* 30, 2 (1983), 323–342. https://doi.org/10.1145/322374.322380

[9] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. 2022. Asynchronous Sessions with Input Races. In *Proceedings of the 13th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES@ETAPS 2022, Munich, Germany, 3rd April 2022 (EPTCS, Vol. 356)*, Marco Carbone and Rumyana Neykova (Eds.). 12–23. https://doi.org/10.4204/EPTCS.356.2

[10] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. 2024. Global Types and Event Structure Semantics for Asynchronous Multiparty Sessions. *Fundam. Informaticae* 192, 1 (2024), 1–75. https://doi.org/10.3233/FI-242188

[11] David Castro-Perez, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. 2019. Distributed programming using role-parametric session types in go: statically-typed endpoint APIs for dynamically-instantiated communication structures. *Proc. ACM Program. Lang.* 3, POPL (2019), 29:1–29:30. https://doi.org/10.1145/3290342

[12] David Castro-Perez and Nobuko Yoshida. 2023. Dynamically Updatable Multiparty Session Protocols: Generating Concurrent Go Code from Unbounded Protocols. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:30. https://doi.org/10.4230/LIPICS.ECOOP.2023.6

[13] Lorenzo Clemente, Slawomir Lasota, and Radoslaw Piórkowski. 2022. Determinisability of register and timed automata. *Log. Methods Comput. Sci.* 18, 2 (2022). https://doi.org/10.46298/LMCS-18(2:9)2022

[14] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. 2015. A Gentle Introduction to Multiparty Asynchronous Session Types. In *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures (Lecture Notes in Computer Science, Vol. 9104)*, Marco Bernardo and Einar Broch Johnsen (Eds.). Springer, 146–178. https://doi.org/10.1007/978-3-319-18941-3_4

[15] Luís Cruz-Filipe and Fabrizio Montesi. 2020. A core model for choreographic programming. *Theor. Comput. Sci.* 802 (2020), 38–66. https://doi.org/10.1016/j.tcs.2019.07.005

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

[17] Francesco Dagnino, Paola Giannini, and Mariangiola Dezani-Ciancaglini. 2021. Deconfined Global Types for Asynchronous Sessions. In *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12717)*, Ferruccio Damiani and Ornela Dardha (Eds.). Springer, 41–60. https://doi.org/10.1007/978-3-030-78142-2_3

[18] Haitao Dan, Robert M. Hierons, and Steve Counsell. 2010. Non-local Choice and Implied Scenarios. In *8th IEEE International Conference on Software Engineering and Formal Methods, SEFM 2010, Pisa, Italy, 13-18 September 2010*, José Luiz Fiadeiro, Stefania Gnesi, and Andrea Maggiolo-Schettini (Eds.). IEEE Computer Society, 53–62. https:

//doi.org/10.1109/SEFM.2010.14

[19] Loris D'Antoni, Tiago Ferreira, Matteo Sammartino, and Alexandra Silva. 2019. Symbolic Register Automata. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11561)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 3–21. https://doi.org/10.1007/978-3-030-25540-4_1

[20] Loris D'Antoni and Margus Veanes. 2017. The Power of Symbolic Automata and Transducers. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10426)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 47–67. https://doi.org/10.1007/978-3-319-63387-9_3

[21] Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. 2021. Resource-Aware Session Types for Digital Contracts. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. IEEE, 1–16. https://doi.org/10.1109/CSF51468.2021.00004

[22] Ankush Das and Frank Pfenning. 2020. Session Types with Arithmetic Refinements. In *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference) (LIPIcs, Vol. 171)*, Igor Konnov and Laura Kovács (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:18. https://doi.org/10.4230/LIPICS.CONCUR.2020.13

[23] Ankush Das and Frank Pfenning. 2022. Rast: A Language for Resource-Aware Session Types. *Log. Methods Comput. Sci.* 18, 1 (2022). https://doi.org/10.46298/LMCS-18(1:9)2022

[24] Jan de Muijnck-Hughes and Wim Vanderbauwhede. 2019. A Typing Discipline for Hardware Interfaces. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 6:1–6:27. https://doi.org/10.4230/LIPIcs.ECOOP.2019.6

[25] Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2015. Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. *Formal Methods Syst. Des.* 46, 3 (2015), 197–225. https://doi.org/10.1007/S10703-014-0218-8

[26] Volker Diekert and Grzegorz Rozenberg (Eds.). 1995. *The Book of Traces*. World Scientific. https://doi.org/10.1142/2563

[27] Javier Esparza and Mogens Nielsen. 1994. Decidability Issues for Petri Nets - a survey. *J. Inf. Process. Cybern.* 30, 3 (1994), 143–160.

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

[29] Alain Finkel and Étienne Lozes. 2023. Synchronizability of Communicating Finite State Machines is not Decidable. *Log. Methods Comput. Sci.* 19, 4 (2023). https://doi.org/10.46298/LMCS-19(4:33)2023

[30] Thomas Gazagnaire, Blaise Genest, Loïc Hélouët, P. S. Thiagarajan, and Shaofa Yang. 2007. Causal Message Sequence Charts. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4703)*, Luís Caires and Vasco Thudichum Vasconcelos (Eds.). Springer, 166–180. https://doi.org/10.1007/978-3-540-74407-8_12

[31] Blaise Genest and Anca Muscholl. 2005. Message Sequence Charts: A Survey. In *Fifth International Conference on Application of Concurrency to System Design (ACSD 2005), 6-9 June 2005, St. Malo, France*. IEEE Computer Society, 2–4. https://doi.org/10.1109/ACSD.2005.25

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

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

[34] Lorenzo Gheri, Ivan Lanese, Neil Sayers, Emilio Tuosto, and Nobuko Yoshida. 2022. Design-By-Contract for Flexible Multiparty Session Protocols. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany (LIPIcs, Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:28. https://doi.org/10.4230/LIPICS.ECOOP.2022.8

[35] Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi, and Pascal Weisenburger. 2021. Multiparty Languages: The Choreographic and Multitier Cases (Pearl). In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:27. https://doi.org/10.4230/LIPIcs.ECOOP.2021.22

[36] Patrice Godefroid and Mihalis Yannakakis. 2013. Analysis of Boolean Programs. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7795)*, Nir Piterman and Scott A. Smolka (Eds.). Springer, 214–229. https://doi.org/10.1007/978-3-642-36742-7_16

[37] Hannah Gommerstadt, Limin Jia, and Frank Pfenning. 2018. Session-Typed Concurrent Contracts. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 771–798. https://doi.org/10.1007/978-3-319-89884-1_27

[38] Dennis Griffith and Elsa L. Gunter. 2013. LiquidPi: Inferrable Dependent Session Types. In *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7871)*, Guillaume Brat, Neha Rungta, and Arnaud Venet (Eds.). Springer, 185–197. https://doi.org/10.1007/978-3-642-38088-4_13

[39] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: session-type based reasoning in separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 6:1–6:30. https://doi.org/10.1145/3371074

[40] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2022. Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic. *Log. Methods Comput. Sci.* 18, 2 (2022). https://doi.org/10.46298/LMCS-18(2:16)2022

[41] Jonas Kastberg Hinrichsen, Jules Jacobs, and Robbert Krebbers. 2024. Multris: Functional Verification of Multiparty Message Passing in Separation Logic. (2024). https://jihgfee.github.io/papers/multris_manuscript.pdf

[42] Andrew K. Hirsch and Deepak Garg. 2021. Pirouette: Higher-Order Typed Functional Choreographies. *CoRR* abs/2111.03484 (2021). arXiv:2111.03484 https://arxiv.org/abs/2111.03484

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

[44] Kohei Honda, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. 2012. Verification of MPI Programs Using Session Types. In *Recent Advances in the Message Passing Interface - 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7490)*, Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra (Eds.). Springer, 291–293. https://doi.org/10.1007/978-3-642-33518-1_37

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

[46] Raymond Hu and Nobuko Yoshida. 2016. Hybrid Session Verification Through Endpoint API Generation. In *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9633)*, Perdita Stevens and Andrzej Wasowski (Eds.). Springer, 401–418. https://doi.org/10.1007/978-3-662-49665-7_24

[47] Raymond Hu and Nobuko Yoshida. 2017. Explicit Connection Actions in Multiparty Session Types. In *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10202)*, Marieke Huisman and Julia Rubin (Eds.). Springer, 116–133. https://doi.org/10.1007/978-3-662-54494-5_7

[48] Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. 2020. Multiparty Session Programming With Global Protocol Combinators. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:30. https://doi.org/10.4230/LIPICS.ECOOP.2020.9

[49] Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. 2024. Deadlock-Free Separation Logic: Linearity Yields Progress for Dependent Higher-Order Message Passing. *Proc. ACM Program. Lang.* 8, POPL (2024), 1385–1417. https://doi.org/10.1145/3632889

[50] Sung-Shik Jongmans and Petra van den Bos. 2022. A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming. In *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13240)*, Ilya Sergey (Ed.). Springer, 520–547. https://doi.org/10.1007/978-3-030-99336-8_19

[51] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20.

https://doi.org/10.1017/S0956796818000151

[52] Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. 2022. Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types (Artifact). *Dagstuhl Artifacts Ser.* 8, 2 (2022), 09:1–09:16. https://doi.org/10.4230/DARTS.8.2.9

[53] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563

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

[55] Elaine Li, Felix Stutz, and Thomas Wies. 2024. Deciding Subtyping for Asynchronous Multiparty Sessions. In *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 14576)*, Stephanie Weirich (Ed.). Springer, 176–205. https://doi.org/10.1007/978-3-031-57262-3_8

[56] Elaine Li, Felix Stutz, Thomas Wies, and Damien Zufferey. 2023. Complete Multiparty Session Type Projection with Automata. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 13966)*, Constantin Enea and Akash Lal (Eds.). Springer, 350–373. https://doi.org/10.1007/978-3-031-37709-9_17

[57] Elaine Li, Felix Stutz, Thomas Wies, and Damien Zufferey. 2023. Complete Multiparty Session Type Projection with Automata. *CoRR* abs/2305.17079 (2023). https://doi.org/10.48550/ARXIV.2305.17079 arXiv:2305.17079

[58] Elaine Li, Felix Stutz, Thomas Wies, and Damien Zufferey. 2025. Characterizing Implementability of Global Protocols with Infinite States and Data. arXiv:2411.05722 [cs.PL] https://arxiv.org/abs/2411.05722

[59] Markus Lohrey. 2003. Realizability of high-level message sequence charts: closing the gaps. *Theor. Comput. Sci.* 309, 1-3 (2003), 529–554. https://doi.org/10.1016/J.TCS.2003.08.002

[60] Rupak Majumdar, Madhavan Mukund, Felix Stutz, and Damien Zufferey. 2021. Generalising Projection in Asynchronous Multiparty Session Types. In *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference (LIPIcs, Vol. 203)*, Serge Haddad and Daniele Varacca (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 35:1–35:24. https://doi.org/10.4230/LIPICS.CONCUR.2021.35

[61] Rupak Majumdar, Madhavan Mukund, Felix Stutz, and Damien Zufferey. 2021. Generalising Projection in Asynchronous Multiparty Session Types. *CoRR* abs/2107.03984 (2021). arXiv:2107.03984 https://arxiv.org/abs/2107.03984

[62] Rupak Majumdar, Marcus Pirron, Nobuko Yoshida, and Damien Zufferey. 2019. Motion Session Types for Robotic Interactions (Brave New Idea Paper). In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom (LIPIcs, Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 28:1–28:27. https://doi.org/10.4230/LIPIcs.ECOOP.2019.28

[63] Rupak Majumdar, Nobuko Yoshida, and Damien Zufferey. 2020. Multiparty motion coordination: from choreographies to robotics programs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 134:1–134:30. https://doi.org/10.1145/3428202

[64] Sjouke Mauw and Michel A. Reniers. 1997. High-level message sequence charts. In *SDL '97 Time for Testing, SDL, MSC and Trends - 8th International SDL Forum, Evry, France, 23-29 September 1997, Proceedings*, Ana R. Cavalli and Amardeo Sarma (Eds.). Elsevier, 291–306.

[65] Fabrizio Montesi. 2023. *Introduction to Choreographies.* Cambridge University Press. https://doi.org/10.1017/9781108981491

[66] Rémi Morin. 2002. Recognizable Sets of Message Sequence Charts. In *STACS 2002, 19th Annual Symposium on Theoretical Aspects of Computer Science, Antibes - Juan les Pins, France, March 14-16, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2285)*, Helmut Alt and Afonso Ferreira (Eds.). Springer, 523–534. https://doi.org/10.1007/3-540-45841-7_43

[67] Madhavan Mukund. 2002. *From Global Specifications to Distributed Implementations.* Springer US, Boston, MA, 19–35. https://doi.org/10.1007/978-1-4757-6656-1_2

[68] Anca Muscholl and Doron A. Peled. 1999. Message Sequence Graphs and Decision Problems on Mazurkiewicz Traces. In *Mathematical Foundations of Computer Science 1999, 24th International Symposium, MFCS'99, Szklarska Poreba, Poland, September 6-10, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1672)*, Miroslaw Kutylowski, Leszek Pacholski, and Tomasz Wierzbicki (Eds.). Springer, 81–91. https://doi.org/10.1007/3-540-48340-3_8

[69] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. 2017. Timed runtime monitoring for multiparty conversations. *Formal Aspects Comput.* 29, 5 (2017), 877–910. https://doi.org/10.1007/S00165-017-0420-8

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

[71] Rumyana Neykova and Nobuko Yoshida. 2017. Multiparty Session Actors. *Log. Methods Comput. Sci.* 13, 1 (2017). https://doi.org/10.23638/LMCS-13(1:17)2017

[72] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. 2012. Multiparty Session C: Safe Parallel Programming with Message Optimisation. In *Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7304)*, Carlo A. Furia and Sebastian Nanz (Eds.). Springer, 202–218. https://doi.org/10.1007/978-3-642-30561-0_15

[73] Xinyu Niu, Nicholas Ng, Tomofumi Yuki, Shaojun Wang, Nobuko Yoshida, and Wayne Luk. 2016. EURECA compilation: Automatic optimisation of cycle-reconfigurable circuits. In *26th International Conference on Field Programmable Logic and Applications, FPL 2016, Lausanne, Switzerland, August 29 - September 2, 2016*, Paolo Ienne, Walid A. Najjar, Jason Helge Anderson, Philip Brisk, and Walter Stechele (Eds.). IEEE, 1–4. https://doi.org/10.1109/FPL.2016.7577359

[74] Abhik Roychoudhury, Ankit Goel, and Bikram Sengupta. 2012. Symbolic Message Sequence Charts. *ACM Trans. Softw. Eng. Methodol.* 21, 2 (2012), 12:1–12:44. https://doi.org/10.1145/2089116.2089122

[75] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. 2017. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 24:1–24:31. https://doi.org/10.4230/LIPICS.ECOOP.2017.24

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

[77] Gan Shen, Shun Kashiwa, and Lindsey Kuper. 2023. HasChor: Functional Choreographic Programming for All (Functional Pearl). *CoRR* abs/2303.00924 (2023). https://doi.org/10.48550/ARXIV.2303.00924 arXiv:2303.00924

[78] Felix Stutz. 2023. Asynchronous Multiparty Session Type Implementability is Decidable - Lessons Learned from Message Sequence Charts. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:31. https://doi.org/10.4230/LIPICS.ECOOP.2023.32

[79] Felix Stutz. 2024. *Implementability of Asynchronous Communication Protocols - The Power of Choice.* Ph. D. Dissertation. Kaiserslautern University of Technology, Germany. https://kluedo.ub.rptu.de/frontdoor/index/index/docId/8077

[80] Peter Thiemann and Vasco T. Vasconcelos. 2020. Label-dependent session types. *Proc. ACM Program. Lang.* 4, POPL (2020), 67:1–67:29. https://doi.org/10.1145/3371135

[81] Bernardo Toninho and Nobuko Yoshida. 2017. Certifying data in multiparty session types. *J. Log. Algebraic Methods Program.* 90 (2017), 61–83. https://doi.org/10.1016/J.JLAMP.2016.11.005

[82] International Telecommunication Union. 1996. *Z.120: Message Sequence Chart.* Technical Report. International Telecommunication Union. https://www.itu.int/rec/T-REC-Z.120

[83] Hiroshi Unno, Tachio Terauchi, Yu Gu, and Eric Koskinen. 2023. Modular Primal-Dual Fixpoint Logic Solving for Temporal Verification. *Proc. ACM Program. Lang.* 7, POPL (2023), 2111–2140. https://doi.org/10.1145/3571265

[84] Margus Veanes and Nikolaj S. Bjørner. 2012. Symbolic Automata: The Toolkit. In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7214)*, Cormac Flanagan and Barbara König (Eds.). Springer, 472–477. https://doi.org/10.1007/978-3-642-28756-5_33

[85] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. 2010. Rex: Symbolic Regular Expression Explorer. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010.* IEEE Computer Society, 498–507. https://doi.org/10.1109/ICST.2010.15

[86] Nobuko Yoshida. 2024. Programming Language Implementations with Multiparty Session Types. In *Active Object Languages: Current Research Trends*, Frank S. de Boer, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, and Eduard Kamburjan (Eds.). Lecture Notes in Computer Science, Vol. 14360. Springer, 147–165. https://doi.org/10.1007/978-3-031-51060-1_6

[87] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. 2013. The Scribble Protocol Language. In *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8358)*, Martín Abadi and Alberto Lluch-Lafuente (Eds.). Springer, 22–41. https://doi.org/10.1007/978-3-319-05119-2_3

[88] Fangyi Zhou. 2024. *Refining Multiparty Session Types.* Ph. D. Dissertation. Imperial College London.

[89] Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2020. Statically verified refinements for multiparty protocols. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 148:1–148:30. https://doi.org/10.1145/3428216

[90] Wieslaw Zielonka. 1987. Notes on Finite Asynchronous Automata. *RAIRO Theor. Informatics Appl.* 21, 2 (1987), 99–135. https://doi.org/10.1051/ITA/1987210200991

Received 2024-10-16; accepted 2025-02-18