


Certified Implementability of Global Multiparty Protocols

Elaine Li  

New York University, New York, USA

Thomas Wies 

New York University, New York, USA

Abstract

Implementability is the decision problem at the heart of top-down approaches to protocol verification. In this paper, we present a mechanization of a recently proposed precise implementability characterization by Li et al. for a large class of protocols that subsumes many existing formalisms in the literature. Our protocols and implementations model asynchronous communication, and can exhibit infinite behavior. We improve upon their pen-and-paper results by unifying distinct formalisms, simplifying existing proof arguments, elaborating on the construction of canonical implementations, and even uncovering a subtle bug in the semantics for infinite words. As a corollary of our mechanization, we show that the original characterization of implementability applies even to protocols with infinitely many participants. We also contribute a reusable library for reasoning about generic communicating state machines. Our mechanization consists of about 15k lines of Rocq code. We believe that our mechanization can provide the foundation for deductively proving the implementability of protocols beyond the reach of prior work, extracting certified implementations for finite protocols, and investigating implementability under alternative asynchronous communication models.

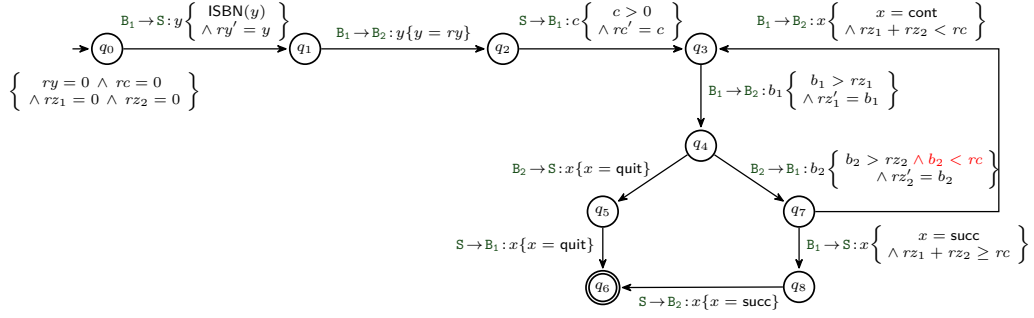
2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Distributed computing models; Theory of computation → Automata over infinite objects

Keywords and phrases Asynchronous protocols, communicating state machines, labeled transition systems, infinite semantics, realizability, multiparty session types, choreographies, deadlock freedom

1 Introduction

Distributed, message-passing protocols are notoriously difficult to implement correctly due to exhibiting potentially infinitely many behaviors, only a handful of which may contain bugs. One salient verification methodology centers on the construct of a *global* protocol, which specifies the behaviors of all participants from the perspective of an omniscient observer. Global protocols enjoy certain desirable correctness properties by design, such as the absence of unspecified message receptions, and deadlock freedom. Top-down verification methodologies use global protocols as a starting point to synthesize correct-by-construction, distributed implementations in an automatic or semi-automatic fashion, that inherit all desirable properties of the global specification. The approach is attractive due to its simplicity and use of automation. It is thus adopted by numerous existing frameworks including multiparty session types [26, 4, 3, 49, 50, 32], choreography automata and choreographic programming [18, 25, 44], and high-level message sequence charts [38, 16, 15, 14, 42, 1, 35, 2, 40, 39, 17].

The decision problem at the heart of all such top-down approaches is *implementability*, which asks whether a given global specification admits a distributed implementation. A distributed implementation is considered admissible if it is deadlock-free and exhibits exactly the same communication behaviors as described by the specification. Implementability can be understood as a meta-correctness property of global specifications.



■ **Figure 1** Non-implementable two-bidder protocol. If one omits the constraint $b_2 < rc$ in the transition from q_4 to q_7 , the protocol is implementable.

To see that implementability is a subtle problem, consider the faulty two-bidder protocol adapted from [33] and shown in Figure 1. The protocol specifies a negotiation between two bidders B_1 and B_2 to split the price c for the purchase of a book y from seller S . A transition $p \rightarrow q: x \{ \phi \}$ in the protocol specifies that p sends value x to q under transition constraint ϕ . A transition constraint expresses conditions on the sent value x in relation to register variables (prefixed with r) as well as updates to these register variables (expressed in terms of primed register variables). In the loop formed by the control states q_3 , q_4 , and q_7 , the two bidders exchange bids b_1 and b_2 until either B_2 aborts the protocol, or the sum of the bids exceeds the books price (stored in rc). Registers rz_1 and rz_2 are used to track the latest bids of each bidder.

Note that the only way in which protocol participants can learn information about the global protocol state is via the sent message values. Consequently, the protocol is not implementable. The reason is that the transition from q_4 to q_7 requires B_2 to choose a bid b_2 that is strictly smaller than rc . However, B_2 cannot know the value of rc because the book's price is never disclosed to B_2 . If the constraint $b_2 < rc$ is dropped from the transition, then the protocol becomes implementable.

The problem of implementability is of both theoretical interest and practical importance. In the high-level message sequence chart literature, implementability is referred to as realizability, and its decidability and complexity have been thoroughly studied for finite state protocols [38, 16, 15, 14, 42]. Frameworks such as multiparty session types [26, 4, 3, 49, 50, 32] and choreographic programming [9, 20, 25, 44] combine implementability checking and synthesis in a single step known as endpoint projection. Unsound implementability checks in these frameworks may result in implementations that exhibit communication errors or deadlocks, whereas incomplete implementability checks undermine their utility.

Despite the fact that the vast majority of existing implementability checks are conservative and do not aim for completeness, multiple unsound implementability checks have been proposed [6, 8, 11, 10, 49], in addition to false claims about the decidability of implementability for various protocol classes [18] that were later refuted.

Mechanization has proven to be an effective way to fortify the correctness of pen-and-paper results. In the domain of process calculi, a mechanization of [29] called HOCORE [37] revealed and subsequently fixed several major flaws in the existing proofs. Proof assistants especially excel at preventing inexhaustive case analysis, which was shown by [13] to be the cause of erroneous prior works claiming the decidability of the realizability and synchronizability problems for systems of asynchronously communicating state machines.

However, all existing works in the intersection of mechanization and protocol imple-

mentability consider restricted implementation models that support only synchronous communication [48, 25] or asynchronous communication with directed choice [7]. Moreover, specifications are restricted to protocols with finitely many participants and completeness is stated relative to projection operators that are themselves incomplete for implementability.

Contributions. In this paper, we present a mechanization of a recently proposed precise implementability characterization for a large class of protocols that subsumes many existing formalisms in the literature [33, 34]. Throughout this paper, we refer to the extended version [34] of [33] containing complete proofs. Our protocols and implementations model asynchronous communication, and can exhibit infinite behavior. Our semantic model of protocols unifies two distinct formalisms from [34] under one general definition, which is capable of expressing syntactic formalisms such as multiparty session types and choreographic programs. We improve upon the results in [34] by simplifying existing proof arguments, elaborating on the construction of canonical implementations, and even uncovering a subtle bug in the semantics for infinite words. As a corollary of our mechanization, we show that the characterization in [34] applies even to protocols with infinitely many participants. We also contribute a reusable library for reasoning about generic communicating state machines, which can serve as a basis for formalizing other theoretical results in concurrency theory.

2 Preliminaries

We introduce basic concepts and notation, and tour the main result from [34], introducing relevant definitions along the way.

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 $u \leq v$, if there exists $w \in \Sigma^\infty$ with $u \cdot w = v$; we denote all prefixes of u with $\text{pref}(u)$. 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 (possibly infinite) set of participants and \mathcal{V} be a (possibly infinite) data domain. We define the set of *synchronous events* $\Gamma_{sync} := \{\mathbf{p} \rightarrow \mathbf{q} : m \mid \mathbf{p}, \mathbf{q} \in \mathcal{P} \text{ and } m \in \mathcal{V}\}$ where $\mathbf{p} \rightarrow \mathbf{q} : m$ denotes a message exchange of m from sender \mathbf{p} to receiver \mathbf{q} . For a participant $\mathbf{p} \in \mathcal{P}$, we define the alphabet $\Gamma_{\mathbf{p}} = \{\mathbf{p} \rightarrow \mathbf{q} : m \mid \mathbf{q} \in \mathcal{P}, m \in \mathcal{V}\} \cup \{\mathbf{q} \rightarrow \mathbf{p} : m \mid \mathbf{q} \in \mathcal{P}, m \in \mathcal{V}\}$, and a homomorphism $\Downarrow_{\Gamma_{\mathbf{p}}}$, where $x \Downarrow_{\Gamma_{\mathbf{p}}} = x$ if $x \in \Gamma_{\mathbf{p}}$ and ε otherwise. For a participant $\mathbf{p} \in \mathcal{P}$, we define the alphabet $\Sigma_{\mathbf{p},!} = \{\mathbf{p} \triangleright \mathbf{q} ! m \mid \mathbf{q} \in \mathcal{P}, m \in \mathcal{V}\}$ of *send* events and the alphabet $\Sigma_{\mathbf{p},?} = \{\mathbf{p} \triangleleft \mathbf{q} ? m \mid \mathbf{q} \in \mathcal{P}, m \in \mathcal{V}\}$ of *receive* events. The event $\mathbf{p} \triangleright \mathbf{q} ! m$ denotes participant \mathbf{p} sending a message m to \mathbf{q} , and $\mathbf{p} \triangleleft \mathbf{q} ? m$ denotes participant \mathbf{p} receiving a message m from \mathbf{q} . We write $\Sigma_{\mathbf{p}} = \Sigma_{\mathbf{p},!} \cup \Sigma_{\mathbf{p},?}$, $\Sigma_! = \bigcup_{\mathbf{p} \in \mathcal{P}} \Sigma_{\mathbf{p},!}$, and $\Sigma_? = \bigcup_{\mathbf{p} \in \mathcal{P}} \Sigma_{\mathbf{p},?}$. Finally, the set of *asynchronous events* is $\Sigma_{async} = \Sigma_! \cup \Sigma_?$. We define a homomorphism to map words from the synchronous alphabet to their asynchronous counterpart, $\text{split}(\mathbf{p} \rightarrow \mathbf{q} : m) := \mathbf{p} \triangleright \mathbf{q} ! m. \mathbf{q} \triangleleft \mathbf{p} ? m$. We say that \mathbf{p} is *active* in $x \in \Sigma_{async}$ if $x \in \Sigma_{\mathbf{p}}$. For each participant $\mathbf{p} \in \mathcal{P}$, we define a homomorphism $\Downarrow_{\Sigma_{\mathbf{p}}}$, where $x \Downarrow_{\Sigma_{\mathbf{p}}} = x$ if $x \in \Sigma_{\mathbf{p}}$ and ε otherwise. We write $\mathcal{V}(w)$ to project the send and receive events in w onto their messages.

The basic building block for specifying global protocols in [34] is a labeled transition system over the synchronous alphabet Γ_{sync} .

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

126 *maximal* if it is either finite and ends in a final state, or is infinite. The language of an LTS
 127 \mathcal{S} , denoted $\mathcal{L}(\mathcal{S})$, is defined as the set of maximal traces. An LTS is *deadlock-free* if every
 128 run is extensible to a maximal run. Given an LTS $\mathcal{S} = (S, \Gamma, T, s_0, F)$ and a state $s \in S$, we
 129 use \mathcal{S}_s to denote the LTS obtained by replacing s_0 with s as the initial state: (S, Γ, T, s, F) .

130 In [34], LTS over Γ_{sync} are constrained with three additional conditions, to yield a
 131 fragment called *global communicating labeled transition systems*, hereafter GCLTS. The three
 132 GCLTS assumptions are sink finality, sender-driven choice, and deadlock freedom. Sink
 133 finality is a purely syntactic condition enforcing that final states have no outgoing transitions.
 134 Sender-driven choice states that from any state, all outgoing transitions are labeled with
 135 events that share a unique sender, and moreover no two transitions are labeled with the
 136 same event. Deadlock freedom requires that every run is extensible to a maximal one.

137 In the remainder of the paper, let $\mathcal{S} = (S, \Gamma_{sync}, T, s_0, F)$ be a protocol satisfying GCLTS
 138 assumptions. The standard LTS semantics of \mathcal{S} yields a set of finite and infinite synchronous
 139 traces. Yet, global protocols describe asynchronous behaviors. The *asynchronous protocol*
 140 *semantics* of \mathcal{S} are defined on top of its LTS semantics, and rely on a notion of *channel*
 141 *compliance*. Channel compliance describes sequences of asynchronous send and receive events
 142 that respect peer-to-peer FIFO ordering, meaning that messages are sent before they are
 143 received, and between every pair of participants, the order of messages received strictly
 144 follows the order of messages sent, with no drops or reordering.

145 **Channel Compliance.** Let $w \in \Sigma_{async}^\infty$. We say that w is *channel-compliant* if for all
 146 prefixes $w' \leq w$, for all $p \neq q \in \mathcal{P}$, $\mathcal{V}(w' \downarrow_{q \triangleleft p ?} _) \leq \mathcal{V}(w' \downarrow_{p \triangleright q !} _)$.

147 **Asynchronous Protocol Semantics.** Let \mathcal{S} be an LTS over Γ_{sync} . The *asynchronous*
 148 semantics of \mathcal{S} , denoted $\mathcal{C}^\sim(\mathcal{S})$, is defined by first mapping synchronous words of \mathcal{S} onto their
 149 asynchronous counterpart using **split**, and then closing the resulting language under an
 150 indistinguishability relation that captures all possible reorderings in an asynchronous, FIFO
 151 network. The semantics for infinite words in \mathcal{S} used in this work differs from that in [34].
 152 We revisit this point in detail in §3.2, and provide the formal definition of \mathcal{S} 's asynchronous
 153 protocol semantics below.

$$\begin{aligned}
 154 \quad \mathcal{C}^\sim(\mathcal{S}) = & \{w' \in \Sigma_{async}^* \mid \exists w \in \Sigma_{async}^*. w \in \mathbf{split}(\mathcal{L}(\mathcal{S})) \wedge w' \text{ is channel-compliant} \\
 155 & \wedge \forall p \in \mathcal{P}. w' \downarrow_{\Sigma_p} = w \downarrow_{\Sigma_p}\} \\
 156 & \cup \{w' \in \Sigma_{async}^\omega \mid \forall v' \leq w'. \exists w \in \Sigma_{async}^\omega. \exists u, u' \in \Sigma_{async}^*. w \in \mathbf{split}(\mathcal{L}(\mathcal{S})) \wedge \\
 157 & u \leq w \wedge v' \cdot u' \text{ is channel-compliant} \wedge \forall p \in \mathcal{P}. (v' \cdot u') \downarrow_{\Sigma_p} = u \downarrow_{\Sigma_p}\} .
 \end{aligned}$$

158 For disambiguation, we refer to $\mathcal{L}(\mathcal{S}) \subseteq \Gamma_{sync}^\omega$ as the *LTS semantics* of \mathcal{S} , and refer to
 159 $\mathcal{C}^\sim(\mathcal{S}) \subseteq \Sigma_{async}^\omega$ as the *protocol semantics* of \mathcal{S} .

160 The implementation model in [34] is *communicating labeled transition systems*, hereafter
 161 CLTS. CLTSs are a generalization of communicating state machines [5] to potentially infinite-
 162 state labeled transition systems for each participant.

163 **Communicating LTS.** $\mathcal{T} = \{\{T_p\}_{p \in \mathcal{P}}\}$ is a *communicating labeled transition system* (CLTS)
 164 over \mathcal{P} and \mathcal{V} if T_p is a deterministic LTS over Σ_p for every $p \in \mathcal{P}$, denoted by $(Q_p, \Sigma_p, \delta_p, q_{0,p}, F_p)$.
 165 Let $\prod_{p \in \mathcal{P}} Q_p$ denote the set of global states and $\mathbf{Chan} = \{(p, q) \mid p, q \in \mathcal{P}, p \neq q\}$ denote
 166 the set of channels. A *configuration* of \mathcal{A} is a pair (\vec{s}, ξ) , where \vec{s} is a global state and
 167 $\xi : \mathbf{Chan} \rightarrow \mathcal{V}^*$ is a mapping from each channel to a sequence of messages. We use \vec{s}_p to
 168 denote the state of p in \vec{s} . The CLTS transition relation, denoted \rightarrow , is defined as follows.

$$\begin{aligned}
 169 \quad \blacksquare \quad (\vec{s}, \xi) & \xrightarrow{p \triangleright q ! m} (\vec{s}', \xi') \text{ if } (\vec{s}_p, p \triangleright q ! m, \vec{s}'_p) \in \delta_p, \vec{s}_r = \vec{s}'_r \text{ for every participant } r \neq p, \xi'(p, q) = \\
 170 \quad & \xi(p, q) \cdot m \text{ and } \xi'(c) = \xi(c) \text{ for every other channel } c \in \mathbf{Chan}.
 \end{aligned}$$

171 $\dashv\vdash (\vec{s}, \xi) \xrightarrow{q \triangleleft p ? m} (\vec{s}', \xi')$ if $(\vec{s}_q, q \triangleleft p ? m, \vec{s}'_q) \in \delta_q$, $\vec{s}_r = \vec{s}'_r$ for every participant $r \neq q$,
 172 $\xi(p, q) = m \cdot \xi'(p, q)$ and $\xi'(c) = \xi(c)$ for every other channel $c \in \text{Chan}$.

173 In the initial configuration (\vec{s}_0, ξ_0) , each participant's state in \vec{s}_0 is the initial state $q_{0,p}$ of A_p ,
 174 and ξ_0 maps each channel to ε . A configuration (\vec{s}, ξ) is *final* iff \vec{s}_p is final for every p and ξ
 175 maps each channel to ε . Runs and traces are defined in the expected way. A run is *maximal*
 176 if either it is finite and ends in a final configuration, or it is infinite. The language $\mathcal{L}(\mathcal{T})$
 177 of the CLTS \mathcal{T} is defined as the set of maximal traces. A configuration (\vec{s}, ξ) is a *deadlock*
 178 if it is not final and has no outgoing transitions. A CLTS is *deadlock-free* if no reachable
 179 configuration is a deadlock.

180 Having defined the model of global protocols and their implementations, we can now
 181 define the implementability problem.

182 **► Definition 1** (Protocol Implementability). *A protocol \mathcal{S} is implementable if there ex-*
 183 *ists a CLTS $\{\{T_p\}_{p \in \mathcal{P}}\}$ such that the following two properties hold: (i) protocol fidelity:*
 184 *$\mathcal{L}(\{\{T_p\}_{p \in \mathcal{P}}\}) = \mathcal{C}^\sim(\mathcal{S})$, and (ii) deadlock freedom: $\{\{T_p\}_{p \in \mathcal{P}}\}$ is deadlock-free. We say that*
 185 *$\{\{T_p\}_{p \in \mathcal{P}}\}$ implements \mathcal{S} .*

186 The key result in [34] is a sound and complete characterization of implementability for
 187 GCLTS with a finite set of participants, formulated as three Coherence Conditions (CC).
 188 In a nutshell, these are 2-hyperproperties stating that from two locally indistinguishable
 189 global protocol states, a participant can either perform a send action that is enabled in
 190 both states (Send Coherence), or perform a receive action that uniquely distinguishes the
 191 two states (Receive Coherence), but cannot choose between performing a send or receive
 192 action (No Mixed Choice). Locally indistinguishable states are captured by the definition of
 193 *simultaneous reachability* for a participant p , denoted $s_0 \xrightarrow[p]{u}^* s_1, s_2$, which says there exist
 194 $w_1, w_2 \in \Gamma_{sync}^*$ such that $s_0 \xrightarrow{w_1}^* s_1 \in T$, $s_0 \xrightarrow{w_2}^* s_2 \in T$ and $w_1 \downarrow_{\Gamma_p} = w_2 \downarrow_{\Gamma_p} = u$. We use
 195 $s_0 \xrightarrow[p]{u}^* s$ to denote participant-based reachability of a single state, i.e. when there exists
 196 $w \in \Gamma_{sync}^*$ such that $s_0 \xrightarrow{w}^* s \in T$ and $w \downarrow_{\Gamma_p} = u$.

197 **► Definition 2** (Coherence Conditions). *A protocol $\mathcal{S} = (S, \Gamma_{sync}, T, s_0, F)$ satisfies Coherence*
 198 *Conditions (CC) when it satisfies:*

- 199 $\dashv\vdash$ Send Coherence: $\forall s_1 \xrightarrow{p \rightarrow q : m} s_2 \in T, s'_1 \in S :$
 200 $(\exists u \in \Gamma_p^*. s_0 \xrightarrow[p]{u}^* s_1, s'_1) \implies (\exists s'_2 \in S. s'_1 \xrightarrow[p]{p \rightarrow q : m}^* s'_2) .$
- 201 $\dashv\vdash$ Receive Coherence: $\forall s_1 \xrightarrow{p \rightarrow q : m} s_2, s'_1 \xrightarrow{r \rightarrow q : m} s'_2 \in T :$
 202 $(r \neq p \wedge \exists u \in \Gamma_q^*. s_0 \xrightarrow[q]{u}^* s_1, s'_1) \implies$
 203 $\neg \exists w \in \text{pref}(\mathcal{L}(\mathcal{S}_{s'_2})). w \downarrow_{\Sigma_q} = \varepsilon \wedge \mathcal{V}(w \downarrow_{p \triangleright q ! _}) = \mathcal{V}(w \downarrow_{q \triangleleft p ? _}) \cdot m .$
- 204 $\dashv\vdash$ No Mixed Choice: $\forall s_1 \xrightarrow{p \rightarrow q : m} s_2, s'_1 \xrightarrow{r \rightarrow p : m} s'_2 \in T :$
 205 $(\exists u \in \Gamma_p^*. s_0 \xrightarrow[p]{u}^* s_1, s'_1) \implies \perp$

206 **► Theorem 3** (Preciseness of Coherence Conditions). *Let \mathcal{S} be a protocol. Then, \mathcal{S} is*
 207 *implementable if and only if it satisfies CC.*

208 **► Note.** In [34] and this work, soundness and completeness are terms defined relative
 209 to the semantic notion of implementability, and thus describe the metacorrectness of the
 210 verification methodology. Soundness of a characterization means that if a protocol satisfies
 211 the characterization, then it is implementable; completeness means that every implementable
 212 protocol satisfies the characterization. The terms are used varyingly elsewhere in the

literature: in [48], soundness and completeness is defined relative to an existing coinductive relation between global and local session types, which is itself incomplete with respect to implementability, meaning that it does not relate every implementable global type with a candidate local type implementation. In turn, the completeness of [48] does not imply completeness with respect to implementability. In many existing works [7, 25], soundness and completeness describes the *correspondence* between global and local behaviors captured by protocol fidelity: soundness means that every global behavior is exhibited by the local implementations, and completeness means that every local implementation behavior is included in the global specification. On the other hand, the flawed type safety proofs of existing multiparty session type frameworks discussed in [43] constitute unsoundness per our definition: projectable global types lead to local types that can deadlock or exhibit communication errors.

To show soundness of CC , the authors [34] first define *canonical implementations*, which serve as the witness to implementability. Canonicity is defined with respect to \mathcal{S} 's protocol semantics, and includes $\mathcal{C}^\sim(\mathcal{S})$ by construction. To show that the canonical implementation's language is included in $\mathcal{C}^\sim(\mathcal{S})$, and moreover that the canonical implementation is deadlock-free, the authors identify a key inductive invariant, which they call intersection set non-emptiness. Intuitively, intersection set non-emptiness says that for every prefix in the canonical CLTS, there exists a finite or infinite maximal run in the protocol that all participants agree on as a possible run from their local perspective, observing only partial information.

To show completeness of CC , the authors proceed via modus tollens, and construct from the negation of each Coherence Condition a trace for which no protocol run is possible. They show that any implementation of the protocol must admit this trace. Thus, either the trace leads to a deadlock, or it leads to a maximal word in the language that does not have a counterpart in $\mathcal{C}^\sim(\mathcal{S})$. Either way, this poses a contradiction to implementability.

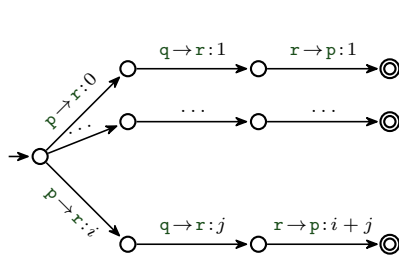
3 Mechanization

We focus our exposition in this section on aspects of the Rocq mechanization that improve upon the pen-and-paper proofs from [34]. In §3.1, we present our purely semantic definition of protocols, which collapses the distinction between GCLTS and symbolic protocols in [34], and can easily encode existing protocol models. In §3.2, we discuss a subtle flaw identified in the infinite word semantics used in [34], its implications on the pen-and-paper proofs, and propose a revised infinite word semantics. As a byproduct, we obtain the generalization from finite to infinite participant sets for free. In §3.3, we present our novel existence proof of canonical implementations. In §3.4, we present a simplification to a key soundness lemma that features nested induction.

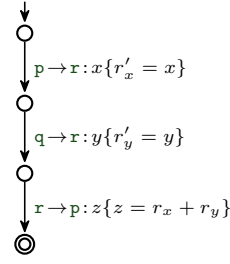
3.1 Protocols as Labeled Transition Systems

In [34], the authors introduce an additional model for finitely representing potentially infinite GCLTS, called *symbolic protocols*. Symbolic protocol states store a set of registers, and transitions are labeled with dependent predicates that can refer to communication variables and register variables, and can thus describe register updates (see e.g. Figure 1). The semantics and implementability of a symbolic protocol is defined in terms of the concrete GCLTS it represents.

For illustration purposes, the GCLTS and symbolic protocol representations of a simple addition protocol between three participants p , q and r are depicted in Figure 2 and Figure 3.



■ **Figure 2** Addition GCLTS.



■ **Figure 3** Addition symbolic protocol.

257 In the protocol, participants p and q send two natural number values x and y to participant
 258 r , who replies to participant p with the sum $z = x + y$, after which the protocol terminates.
 259 [34] thus extends the Coherence Conditions to a set of Symbolic Coherence Conditions
 260 for algorithmically checking implementability of symbolic protocols, as well as investigating
 261 complexity of various decidable symbolic protocol fragments.

262 Thanks to Rocq’s type universe, we unify the two disparate definitions under a single formal
 263 definition in our mechanization, which represents protocols simply as an LTS parametric in
 264 a state and alphabet type, containing a transition relation, an initial state, and a final state
 265 relation.

```

266 Record LTS {A: Type} :=
267   mkLTS { transition: State -> A -> State -> Prop;
268           s0: State;
269           final: State -> Prop; }.
270
271

```

272 We define LTS semantics using an inductive relation to represent reachability, lists
 273 to represent finite traces, and streams to represent infinite traces. Despite the apparent
 274 inconvenience imposed by the type-level distinction between finite and infinite words, we will
 275 see in §3.4 that we can greatly delay the acknowledgement of this distinction in key proofs,
 276 and moreover, that doing so simplifies the existing pen-and-paper proofs.

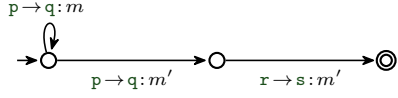
277 3.2 Infinite Protocol Semantics

278 In this section, we examine asynchronous protocol semantics for infinite words.

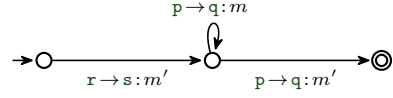
279 As mentioned in §2, the protocol semantics of \mathcal{S} is defined in steps: we begin with the LTS
 280 semantics of \mathcal{S} , then apply a homomorphism `split` to obtain a set of asynchronous words
 281 that remain “synchronously ordered”, i.e. matching send and receive events are adjacent to
 282 each other. In an asynchronous network with peer-to-peer, FIFO channels, certain events
 283 can be reordered, and are thus considered independent. For example, the synchronous trace
 284 $p \rightarrow q: m \cdot r \rightarrow s: m$ yields the asynchronous trace $u_1 = p \triangleright q!m \cdot r \triangleright s!m \cdot s \triangleleft r?m \cdot q \triangleleft p?m$, as
 285 well as $u_2 = r \triangleright s!m \cdot p \triangleright q!m \cdot s \triangleleft r?m \cdot q \triangleleft p?m$, in which the independent sends by p and r
 286 are reordered.

287 We call two words $w, w' \in \Sigma_{\text{async}}^\omega$ *indistinguishable* when any asynchronous implementation
 288 recognizing one word necessarily recognizes the other. Note that indistinguishability is specific
 289 to the assumed communication architecture: two words that are indistinguishable in a peer-
 290 to-peer FIFO setting may not be in a mailbox setting.

291 Allowing the semantics of global protocols to selectively exclude indistinguishable be-
 292 haviors, e.g. by including u_1 but excluding u_2 , would render protocols spuriously non-
 293 implementable. Thus, we desire for our protocol semantics to be closed under this notion of



■ **Figure 4** Example infinite protocol \mathcal{S}_{inf} .



■ **Figure 5** Example infinite protocol \mathcal{S}'_{inf} .

indistinguishability. For finite words, the indistinguishability relation is intuitive to formalize. Prior works that give language-theoretic semantics to session types, such as [36], define indistinguishability in terms of a binary relation on asynchronous events capturing when they can be reordered¹, and a notion of *channel compliance* that captures valid traces with respect to peer-to-peer, FIFO semantics. In the message sequence chart literature, linearizations are required to satisfy the union of per-participant total orders and the send-before-receive partial order on events, coinciding with the definition from [36]. A key observation is that in pairs of indistinguishable finite words, the sequence of events for each participant is identical. Thus, given an asynchronous implementation that recognizes $w \in \Sigma_{async}^\omega$, to show that the same asynchronous implementation recognizes w' indistinguishable from w , we do not need to know more about each participant's local implementation beyond the fact that it accepts $w \downarrow_{\Sigma_p}$, which is given from the fact that the implementation as a whole recognizes w .

For infinite words, however, indistinguishability can no longer be defined purely alphabetically. Consider the pair of infinite words $v_1 = p \triangleright q!m^\omega$ and $v_2 = r \triangleright s!m \cdot p \triangleright q!m^\omega$. Are v_1 and v_2 indistinguishable? On our previous notion of indistinguishability, the answer is unfortunately, no. The fact that v_1 is a trace of an arbitrary asynchronous implementation gives us no information about the local implementation of participant r , yet to show that v_2 is also a trace of said implementation, we need to additionally know that r 's local implementation admits the trace $r \triangleright s!m$. This discrepancy arises from the fact that infinite traces in an asynchronous implementation can infinitely reorder independent events, in this case every occurrence of $p \triangleright q!m$ with $r \triangleright s!m$, achieving the effect of indefinitely postponing $r \triangleright s!m$.

Equipped with an understanding of the importance of indistinguishability-closed global semantics, we revisit the definition of infinite protocol semantics in [34]:

$$\begin{aligned} \mathcal{C}^\sim(\mathcal{S})^\omega &= \{w' \in \Sigma_{async}^\omega \mid \exists w \in \Sigma_{async}^\omega. w \in \mathbf{split}(\mathcal{L}(\mathcal{S})) \wedge \forall v' \leq w'. \exists u, u' \in \Sigma_{async}^* \\ &\quad v' \cdot u' \text{ is channel-compliant} \wedge u \leq w \wedge \forall p \in \mathcal{P}. (v' \cdot u') \downarrow_{\Sigma_p} = u \downarrow_{\Sigma_p}\} . \end{aligned}$$

We show via counterexample that $\mathcal{C}^\sim(\mathcal{S})^\omega$ is not indistinguishability-closed. Consider the simple protocol depicted in Figure 4, involving four participants p, q, r, s and two message values m, m' . As per [34], $\mathcal{C}^\sim(\mathcal{S}_{inf})$ does not include the infinite word $r \triangleright s!m' \cdot p \triangleright q!m^\omega$. In contrast, $\mathcal{C}^\sim(\mathcal{S}'_{inf})$, whose protocol is obtained by a simple state renaming of Figure 4 and is depicted in Figure 5, does include $r \triangleright s!m' \cdot p \triangleright q!m^\omega$.

Before proposing a revised infinite word semantics that resolves this discrepancy, we discuss the implications of this counterexample on the results from [34]. It is easy to verify that \mathcal{S}_{inf} is a GCLTS and satisfies *CC*. However, \mathcal{S}_{inf} is not implementable: there exists no CLTS that recognizes the finite word $p \triangleright q!m^n \cdot p \triangleright q!m' \cdot r \triangleright s!m'$ for all values of $n \in \mathbb{N}$ yet does not recognize the infinite word $r \triangleright s!m \cdot p \triangleright q!m^\omega$. This contradicts the soundness of the Coherence Conditions as stated in [34]. The error lies in the case for infinite words in the proof of [34, Lemma 4.9], which concludes from the fact that every prefix of an infinite

¹ We identify a minor erratum in the original formulation of the indistinguishability relation [36] used in later works [45, 32, 31]: cases (3) and (4) are not symmetric, and thus the relation is not an equivalence relation as claimed.

word w in the canonical implementation has a non-empty intersection run set $I(w)$, that $w \in \mathcal{C}^\sim(\mathcal{S})^\omega$. To show that $w \in \mathcal{C}^\sim(\mathcal{S})^\omega$, one needs to find a witness infinite run ρ in \mathcal{S} , such that for every prefix $v' \leq w$, there exists an extension u' and a prefix of rho ρ'_v such that for all participants, the events prescribed by ρ'_v and $v' \cdot u'$ are identical. To show the existence of such a run, the authors appeal to König's Lemma, and argue that in a finitely-branching infinite tree containing possible run prefixes for every prefix of w' , there exists a ray representing an infinite run. This argument appears inherited from earlier works that deal with finite, multiparty session types [36, 32]. We discover that not only is König's Lemma not applicable in the infinite setting of [34] where GCLTS states can have infinitely many transitions, the existence of a ray is insufficient to prove membership of w in $\mathcal{C}^\sim(\mathcal{S})^\omega$. The latter implies that the proof using König's Lemma in both prior works [36, 32] is flawed: indeed, \mathcal{S}'_{inf} is expressible in the multiparty session type fragments defined in these works that assume finitely many participants, states and transitions. The gap in the reasoning lies in showing that the infinite run obtained from König's Lemma is indeed a suitable existential witness required by the infinite protocol semantics. In the infinite tree constructed for \mathcal{S}_{inf} and word $\mathbf{r} \triangleright \mathbf{s}!m' \cdot \mathbf{p} \triangleright \mathbf{q}!m^\omega$, the prefix $\mathbf{r} \triangleright \mathbf{s}!m'$ contributes a vertex labeled with the run prefix $\mathbf{p} \rightarrow \mathbf{q} : m \cdot \mathbf{p} \rightarrow \mathbf{q} : m' \cdot \mathbf{r} \rightarrow \mathbf{s} : m'$. Subsequent prefixes of the form $\mathbf{r} \triangleright \mathbf{s}!m' \cdot \mathbf{p} \triangleright \mathbf{q}!m^n$ contribute vertices labeled with run prefixes $\mathbf{p} \rightarrow \mathbf{q} : m^n \cdot \mathbf{p} \rightarrow \mathbf{q} : m' \cdot \mathbf{r} \rightarrow \mathbf{s} : m'$. A ray exists in this finite-degree, infinite tree representing the run $\mathbf{p} \rightarrow \mathbf{q} : m^\omega$. This is clearly an infinite run in \mathcal{S}_{inf} , but unfortunately does not satisfy the conditions required to show membership of w' in $\mathcal{C}^\sim(\mathcal{S}_{inf})$: for prefix $\mathbf{r} \triangleright \mathbf{s}!m'$ of w , there exists no prefix of $\mathbf{p} \rightarrow \mathbf{q} : m^\omega$ that matches \mathbf{r} 's events.

Fortunately, the flawed infinite word semantics from [34] can easily be amended to accurately reflect the desired, indistinguishability-closed semantics. Our revised infinite word semantics is as follows:

$$\mathcal{C}^\sim_{alt}(\mathcal{S})^\omega = \{w' \in \Sigma_{async}^\omega \mid \forall v' \leq w'. \exists \rho \in \Gamma_{sync}^*, u' \in \Sigma_{async}^*. \rho \in \text{pref}(\mathcal{L}(\mathcal{S})) \wedge v' \cdot u' \text{ is channel-compliant} \wedge \forall \mathbf{p} \in \mathcal{P}. (v' \cdot u') \Downarrow_{\Sigma_{\mathbf{p}}} = \mathbf{split}(\rho) \Downarrow_{\Sigma_{\mathbf{p}}}\} .$$

$\mathcal{C}^\sim_{alt}(\mathcal{S})^\omega$ swaps the first two quantifiers in the original definition, and weakens the requirement that w come from an infinite run to the requirement that w come from a finite run prefix (that could be part of a finite or infinite maximal run in \mathcal{S}). We hypothesize that this revised condition faithfully represents what prior works intended to capture with their infinite protocol semantics. It also more closely matches simulation-based notions of trace equivalence, for example in [50]. This is further evidenced by the fact that the requisite changes to the overall proof were minimal: the flawed König's Lemma argument could simply be omitted in favor of appealing directly to the intersection set non-emptiness inductive invariant, and the completeness proof remained largely unchanged. The latter is due to the fact that for any infinite word w , $w \in \mathcal{C}^\sim(\mathcal{S})^\omega \implies w \in \mathcal{C}^\sim_{alt}(\mathcal{S})^\omega$.

3.3 Constructing Canonical Implementations

Showing that a global protocol is implementable amounts to finding a witness CLTS that implements it. The soundness proof of CC in [34] chooses a particular CLTS as witness, referred to as the *canonical implementation*. The canonicity of an implementation is defined as follows:²

² Note that in [34], the notation $\mathcal{L}(\mathcal{S})$ is overloaded to denote $\mathcal{C}^\sim(\mathcal{S})$.

► **Definition 4** (Canonical implementations [34]). A CLTS $\{\{T_p\}_{p \in \mathcal{P}}\}$ is a canonical implementation for a protocol $\mathcal{S} = (S, \Gamma_{\text{sync}}, T, s_0, F)$ if for every $p \in \mathcal{P}$, T_p satisfies:

(i) $\forall w \in \Sigma_p^*. w \in \mathcal{C}^\sim(T_p) \Leftrightarrow w \in \mathcal{L}(\mathcal{S}) \downarrow_{\Sigma_p}$, and (ii) $\text{pref}(\mathcal{L}(T_p)) = \text{pref}(\mathcal{C}^\sim(\mathcal{S}) \downarrow_{\Sigma_p})$.

In [34], the existence of a canonical implementation for any protocol is assumed. Formally proving the existence of canonical implementations in our mechanization requires constructing an explicit, albeit non-constructive, witness CLTS. The construction is conceptually straightforward; nonetheless, we illustrate key steps here as it is novel to our mechanization.

We begin by observing that because canonicity is defined on a per-participant basis, and with respect to an LTS that is deadlock-free, the definition can be weakened to use the LTS semantics of \mathcal{S} rather than its protocol semantics. The weaker definition avoids reasoning about asynchronous reorderings and channel compliance, and is formalized in Rocq as follows.

In the Rocq definitions below, S is a protocol of type `LTS SyncAlphabet State`, and p is a participant. We choose `State -> Prop` for the state type of local implementations, so S_p is an LTS of type `LTS AsyncAlphabet (State -> Prop)`.

```

Definition canonical_implementation_local_naive S p S_p :=
  (forall w:FinAsyncWord, is_finite_word S_p w ->
    exists w':FinSyncWord, is_finite_word S w' /\ wproj (split w') p = w)
  /\
  (forall w:FinSyncWord, is_finite_word S w ->
    is_finite_word S_p (wproj (split w) p))
  /\
  (forall w:FinAsyncWord, is_trace S_p w ->
    exists w':FinSyncWord, is_trace S w' /\ wproj (split w') p = w)
  /\
  (forall w:FinSyncWord, is_trace S w ->
    is_trace S_p (wproj (split w) p)).

```

The four conjuncts correspond to four inclusions that altogether define the two equalities in Definition 4, and need to be stated separately due to the type mismatch between finite and infinite words.

Our construction for each participant's local implementation can be expressed simply as a composition of two purely automata-theoretic operations: applying the homomorphism \downarrow_{Σ_p} for each participant, followed by determinization. This coincides with the subset construction automaton as defined in [32], whose name we borrow in our definitions. Formally, for each participant $p \in \mathcal{P}$, the result of the second step is an LTS over $\Sigma_p \cup \{\epsilon\}$. To avoid introducing this compounded alphabet and reasoning about identity elements, we define both operations declaratively in one shot, to obtain a local LTS over the alphabet `AsyncAlphabet`, whose states are of type `State -> Prop`, representing subsets of Q .

The initial state is defined relationally as the set of all states reachable on ϵ from s_0 in \mathcal{S} . States in the subset construction are the set of non-empty subsets of states in \mathcal{S} . Final states are defined relationally as sets of states containing at least one final state from \mathcal{S} .

```

Definition initial_subset_construction_state S p :=
  fun s => exists (w : list SyncAlphabet), lts.Reachable S (s0 S) w s
  /\ wproj (split w) p = [].

Definition subset_construction_state S p :=
  fun lstate => exists (s : State), lstate s.

Definition final_subset_construction_state S p :=
  fun lstate => subset_construction_state S p lstate /\

```

```

425 exists (s : State), lstate s /\ final S s.
426

```

427 The transition relation describes triples (ls, a, ls') where ls is a pre-state in the subset
 428 construction, a is an asynchronous alphabet symbol in p 's restricted alphabet, and ls' is
 429 a post-state. The relation states that ls' contains all states from \mathcal{S} that are either an
 430 immediate post-state of some state s in ls , or is ϵ -reachable from an immediate post-state.

```

431
432 Definition subset_construction_transition_relation S p :=
433   fun lstate1 a lstate2 => is_active p a
434   /\ subset_construction_state S p lstate1
435   /\ subset_construction_state S p lstate2
436   /\ forall (s':State), lstate2 s' <->
437     (exists (s:State), lstate1 s /\ transition S s (async_to_sync a) s')
438   /\
439     (exists (s s_inter:State), lstate1 s /\
440       transition S s (async_to_sync a) s_inter /\
441       exists (v_epsilon:list SyncAlphabet),
442         lts.Reachable S s_inter v_epsilon s' /\
443         wproj (split v_epsilon) p = []).
444

```

445 The former two conjuncts are implied by the latter two conjuncts together with the
 446 definition of final states in the subset construction. The latter two conjuncts state that
 447 every asynchronous trace in a participant's canonical local implementation corresponds to a
 448 synchronous trace in \mathcal{S} , and every synchronous trace in \mathcal{S} corresponds to an asynchronous
 449 trace in the participant's canonical local implementation.

450 Unfortunately, these two properties are not themselves inductive: in both cases, the
 451 induction hypothesis is not strong enough to show that the respective traces can be extended.
 452 We state and prove two inductive invariants that explicitly quantify over states of \mathcal{S} in a
 453 participant's canonical local implementation S_p , and weaken them to obtain the third and
 454 fourth conjuncts. The strengthened inductive properties respectively state that for every
 455 reachable state ls on some asynchronous word w in S_p , for every global state s in ls , one
 456 can find a corresponding synchronous word w' such that w' and w agree on participant
 457 p 's events, and \mathcal{S} reaches s on w' ; conversely, for every reachable global state s on some
 458 synchronous word w in \mathcal{S} , one can find a corresponding local state ls' and asynchronous
 459 word w' such that w' and w agree on participant p 's events, and S_p reaches ls' on w' .

460 Finally, we define the canonical CLTS by mapping each participant to their subset
 461 construction. To show that the map thus defined is indeed a CLTS, we additionally need
 462 to prove that each local implementation is deterministic, and moreover operates on its own
 463 restricted alphabet. Both proofs are straightforward by definition of the subset construction;
 464 our proof of determinism uses the axioms of functional and propositional extensionality from
 465 Rocq's Logic library to establish the equality of local states of type `State` \rightarrow `Prop`. We
 466 conclude with the existence lemma:

```

467
468 Lemma canonical_implementation_exists :
469   forall (S : @LTS SyncAlphabet State),
470     deadlock_free S ->
471     exists (T : CLTS),
472     @canonical_implementation (State -> Prop) S T.
473

```

474 3.4 Simplification of Soundness

475 The core argument for soundness in [34] relies on proving the following inductive invariant:

Let \mathcal{S} be a protocol satisfying CC , and let $\{\{T_p\}_{p \in \mathcal{P}}\}$ be a canonical CLTS for \mathcal{S} . Let w be a trace of $\{\{T_p\}_{p \in \mathcal{P}}\}$. Then, $I(w) \neq \emptyset$.

The set $I(w)$ contains finite or infinite maximal runs in \mathcal{S} that are *possible* with respect to the trace w . Formally, $\rho \in I(w)$ means that for every participant $p \in \mathcal{P}$, $w \downarrow_{\Sigma_p} \leq \text{split}(\rho) \downarrow_{\Sigma_p}$, i.e. each participant's local events in w agree with what ρ prescribes. The proof proceeds by induction on the length of w , with case analysis in the inductive step on whether the next event is a send or receive event.

The non-emptiness of $I(w)$ amounts to an existential quantification over a disjunction. In our mechanization, however, due to the type-level distinction between finite and infinite runs, this property takes the form of a disjunction over existentials:

```

Definition I_set_non_empty (S: LTS) (w: FinAsyncWord) :=
  (exists (run: FinSyncWord), finite_possible_run S run w)
  \/
  (exists (run: InfSyncWord), infinite_possible_run S run w).

```

Although the soundness arguments from [34] are mechanizable using this definition of intersection set non-emptiness, doing so would involve repetitive reasoning to deal with finite and infinite runs separately that does not shed additional insight on the problem. We instead prove that every canonical CLTS trace has a possible run *prefix*. Our new inductive invariant factors out the distinction between finite and infinite runs, and is additionally more expressive than its pen-and-paper counterpart: it makes explicit the construction of a possible run prefix for wx from one for w . When x is a receive event, our lemma states that the exact same run prefix can be reused. When x is a send event, a run prefix can be constructed incrementally by processing w in increasing length order, and appealing to CC to incrementally extend a prefix of the possible run prefix for wx .

We focus the exposition below on our simplified proof for the inductive step when x is a send event. Lemma 4.16 in [34] is stated as follows:

► **Lemma 5** (Send events preserve run prefixes). *Let \mathcal{S} be a protocol satisfying CC and $\{\{T_p\}_{p \in \mathcal{P}}\}$ be a canonical implementation for \mathcal{S} . Let wx be a trace of $\{\{T_p\}_{p \in \mathcal{P}}\}$ such that $x \in \Sigma_p$, for some $p \in \mathcal{P}$. 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 $\alpha \cdot s_{pre} \leq \rho'$.*

The *unique splitting* of a run for a participant with respect to a trace is the largest prefix of the run that matches the participant's actions in the trace, formalized as follows:

```

Definition is_alpha (run alpha: FinSyncWord) (w: FinAsyncWord) p :=
  prefix alpha run /\ wproj w p = wproj (split alpha) p /\
  (forall (u: FinSyncWord), wproj w p = wproj (split u) p ->
    prefix u run -> prefix u alpha).

```

For example, the unique splitting of run $\rho = p \rightarrow q: m \cdot r \rightarrow s: m \cdot r \rightarrow q: m \cdot q \rightarrow p: m$ for participant p with respect to trace $u = p \triangleright q!m \cdot r \triangleright s!m \cdot r \triangleright q!m$ is $p \rightarrow q: m \cdot r \rightarrow s: m \cdot r \rightarrow q: m$, because p has only completed the first event prescribed by ρ in u , namely sending m to r , but has not completed the second event, namely receiving m from q . Because the two synchronous events in between these two events in ρ do not concern p , they are included in the *largest* prefix. If a run disagrees with a trace on some participant's actions, the unique splitting is ϵ , for example ρ 's unique splitting for participant r with respect to trace $r \triangleright s!m'$.

Our adapted formalization of Lemma 5 is thus stated as follows:

```

524
525 Lemma send_preserves_run_prefixes_finite :
526   forall S T w x rho_fin alpha,
527     GCLTS S -> NMC S -> SCC S -> RCC S ->
528     canonical_implementation S T ->
529     is_clts_trace T w -> is_clts_trace T (w ++ [x]) -> is_snd x ->
530     possible_run_prefix S rho_fin w ->
531     is_alpha rho alpha w (sender_async x) ->
532     exists (rho' : FinSyncWord),
533     prefix alpha rho' /\ possible_run_prefix S rho' (w ++ [x]).

```

535 The proof of Lemma 4.16 in [34] relies on a nested induction argument. We illustrate
 536 the key steps in order to elucidate the structure of the nested induction and explain our
 537 simplified proof. From the induction hypothesis, we are granted a canonical CLTS trace
 538 w and a possible run prefix for w . Let the send extension to w be $x = \text{Snd } p \ q \ m$. We can
 539 then define the largest prefix of ρ matching w for participant p , and because the premise
 540 grants that $\alpha \neq \rho$, there must exist a next action prescribed by ρ for p , which we
 541 denote l . As a reminder, since ρ is a run of the global protocol, which is an LTS over the
 542 synchronous alphabet, l is a synchronous alphabet symbol. By the induction hypothesis,
 543 ρ is compliant with all participants:

```

544   forall (p : participant), prefix (wproj w p) (wproj (split rho) p)

```

545 The induction step asks to construct an existential witness for a new possible run prefix,
 546 ρ' , that is compliant with wx . In the case that $l = \text{Event } p \ q \ m$, we can directly reuse
 547 ρ as our witness, and the three conjuncts required of ρ' are trivially satisfied when $\rho' = \rho$.
 548 When this is not the case, we must construct a different witness. We first appeal
 549 to Send Coherence Condition to show that we can find a different continuation from α
 550 that agrees with x , in other words, $l' = [\text{Event } p \ q \ m]$ and $\alpha ++ l'$ is a run in the
 551 global protocol.

552 With this extension and removal of the original suffix from ρ , however, we are left only
 553 with a guarantee about p 's compliance:

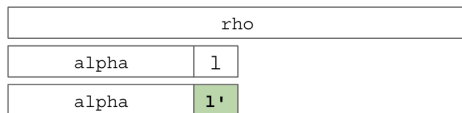
```

554   prefix (wproj (w ++ [x]) p) (wproj (split (alpha ++ [Event p q m])) p)

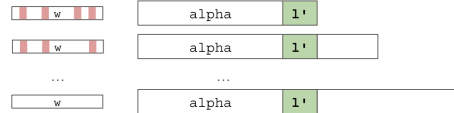
```

555 In the case that all of the actions in w were already contained in α , we can use α
 556 $++ [l']$ directly as our witness for ρ' . However, in the case that some of w 's actions were
 557 contained in the now removed suffix, it is no longer true that all participants are compliant
 558 with $\alpha ++ [l']$. Therefore, the next step of the proof involves restoring a suffix that is
 559 “long enough” to contain all of the actions that were originally in w .

560 The argument for suffix restoration in [34] is algorithmic in nature, and is captured
 561 by the pseudocode in Algorithm 1. The algorithm initializes the candidate run ρ_c as $\alpha \cdot l'$
 562 appended to an arbitrary run suffix β to form a maximal run. The outer while loop then
 563 “fixes” disagreements between w and the current candidate run ρ_c one symbol at a time,
 564 updating ρ_c after each fix. Termination is guaranteed by the fact that w has finite length



■ Figure 6 Induction hypothesis.



■ Figure 7 Inner induction hypothesis.

and that each event in w is fixed at most once. The outer while loop invariant relates ρ'_c with ρ_c , and guarantees that the largest common prefix shared by ρ'_c and ρ_c between each loop iteration is strictly increasing. Because the initial candidate run is picked such that it includes $\alpha \cdot \mathbf{l}'$ as a prefix, and the common prefix between runs can only get longer, it holds by transitivity that when the while loop terminates, the final candidate run must have $\alpha \cdot \mathbf{l}'$ as a prefix, and furthermore is compliant with all events in w .

■ **Algorithm 1** Algorithmic representation of Lemma 4.16 [34]

```

▷ Let  $\rho_c$  be  $\alpha \cdot \mathbf{l}' \cdot \beta$ , where  $\beta$  is an arbitrary maximal suffix
 $\rho_c \leftarrow \alpha \cdot \mathbf{l}' \cdot \beta$ 
while  $\neg(\forall p \in \mathcal{P}. w \downarrow_{\Sigma_p} \leq \text{split}(\rho_c) \downarrow_{\Sigma_p})$  do
  ▷  $i$  is the index of the earliest disagreeing event in  $\rho_c$ 
   $i \leftarrow \text{length}(\rho_c)$ 
  ▷  $j$  iterates over all prefixes of  $w$ 
   $j \leftarrow 0$ 
  for  $j \in \{0..length(w)\}$  do
     $k \leftarrow \max\{k' \mid \forall p \in \mathcal{P}. \text{split}(\rho_c[0..k'-1]) \downarrow_{\Sigma_p} \leq w[0..j] \downarrow_{\Sigma_p}\}$ 
    if  $k < i$  then
       $i \leftarrow k$ 
   $j \leftarrow j + 1$ 
  ▷  $y$  is the earliest disagreeing event in  $\rho_c$ 
   $y \leftarrow \text{split}(\rho_c)[i]$ 
  ▷  $y'$  is obtained from SCC to no longer disagree with  $w$ 
   $\rho_c \leftarrow \rho_c[0..i-1] \cdot y'$ 

```

Formalizing the above algorithm in addition to its loop invariants would require a custom inductive predicate that relates the candidate run with disagreeing events in w . The fact that the loop invariant depends on both the current and previous candidate run introduces significant additional complexity.

We find a weaker inductive invariant that eliminates this dependency: it suffices to show that $\alpha \cdot \mathbf{l}' \leq \rho_c$ remains a prefix of the candidate run. This holds trivially upon entry to the while loop, and is preserved by each iteration from the fact that α comes from the original ρ that is compliant with w , and thus no events in w can disagree with events in α .

In our new inductive invariant, $\alpha \cdot \mathbf{l}'$ can now be treated as a constant. We convert the algorithmic reasoning in [34] to the following inner induction hypothesis:

```

H_inner: forall (w': FinAsyncWord), prefix w' w ->
  (exists (beta': FinSyncWord),
    possible_run_prefix S (alpha ++ [Event p q m] ++ beta') w').

```

We prove `H_inner` directly by induction on prefixes of w using `rev_ind` from the standard `List` library. Figure 7 visualizes the simplified inductive argument: the red symbols in w depict disagreeing events in w as a result of removing the suffix from ρ .

To prove `send_preserves_run_prefixes_finite`, we needed to consider cases glossed over in the pen-and-paper proof from [34], and in some case develop arguments from scratch. For example, in the special case when `alpha` is a possible run prefix for participant p for trace w , but prescribes exactly as many events as are in w , we needed to show that either w is a maximal CLTS trace, or a different `alpha` can be found which prescribes more events, by appealing to the sink-finality of \mathcal{S} .

4 Related Work

We refer the reader to [34] for a detailed discussion of related work on multiparty session types and similar formalisms. In the following, we instead focus our attention on comparing with prior efforts to mechanize such formalisms.

Most closely related to our effort are [7] and [48]. Zooid [7] is a mechanized domain-specific language for specifying and implementing asynchronous multiparty session types. [48] mechanizes the soundness and completeness proofs for the projection operator for synchronous multiparty session types proposed in [19]. A key conceptual difference is that our proofs follow a semantic argument grounded in formal language theory whereas both [7] and [48] follow more standard syntactic arguments. More fundamentally, the class of protocol specifications considered in this paper generalizes that of [7, 48] along several dimensions: [48] considers synchronous rather than asynchronous communication and in both works, internal choice syntactically disallows a sender from choosing among multiple receivers (like in state q_4 of Figure 1). Moreover, both papers restrict specifications to finitely many participants and states, and abstract message values in terms of simple types without data refinements. Finally, the notion of completeness considered in [48] is defined relative to the coinductive definition of endpoint projection introduced in [19]. The latter is itself incomplete for our semantically defined notion of implementability. The end-point projection of [7] is likewise incomplete (for our notion of completeness; see the note in Section 2).

Pirouette [25] introduces a language of *functional choreographies* that are converted to a distributed implementation via endpoint projection. The language supports session delegation and higher-order functions, neither of which we include in our model of GCLTS. However, functional choreographies are much more restricted in their distributed behavior than the protocols in our model: communication is synchronous and all participants must remain in lock step. The latter is enforced by requiring that the programmer inserts potentially redundant synchronization messages into the choreography. A proof that the implementations obtained by projection are deadlock-free has been mechanized in Rocq. Similar to [48], the completeness theorem is stated relative to completeness of syntactic projection rather than semantic implementability.

There is a large number of other recent mechanization efforts for session type languages [21, 22, 28, 23, 47, 41, 24, 27, 46, 12]. However, these focus on the formalization of language semantics, compiler correctness, or on proving soundness of session type systems that check implementations against *local* types. The latter describe the behavior of individual participants or communication channels and may be obtained by prior endpoint projection from a global type or specified directly by the programmer. We therefore consider these efforts orthogonal to our work.

5 Conclusion

We summarize the mechanization effort in numbers below, briefly discuss our Rocq user experience, and conclude with a discussion of future directions. Our mechanization is available at <https://zenodo.org/records/15760397> [30].

Contents	LOC
Standard library	0.7K
Message alphabet	2.2K
LTS, CLTS	2.6K
Global protocol	3.3K
Coherence conditions	0.2K
Soundness	5.1K
Completeness	1.3K
Total	15K

Our formal language-theoretic treatment of message-passing concurrency meant that definitions were straightforward to state, and much of the reasoning is equational in nature. Our definition of finite words as lists enabled us to lean heavily on the Rocq Standard Library in addition to the `stdpp.list`. We found significantly less library support for reasoning about streams however, and thus the standard library supplement required for our mechanization primarily consists of lemmas about streams. In addition to the standard induction principles for natural numbers and lists, many proofs relied on alternative induction principles such as strong induction on natural numbers (`lt_wf_ind`) and reverse induction on lists (`rev_ind`).

On the basis of our observation that [33, 34]’s result does not rely on a finite number of participants, it would be interesting to model and prove implementability of protocols with expressive features such as channel creation and delegation that are elided in much of the literature due to their complexity. It would also be interesting to explore using our semantic definition of protocols to model and prove implementability of existing Rocq protocol formalisms, such as [25, 7]. We further hope to use our mechanization to synthesize certified implementations for finite protocols, and investigate implementability under other asynchronous communication architectures. As shown in [32], synthesis for finite protocols amounts to a direct subset construction. As discussed in [33], symbolic, infinite-state automata fragments which admit generalized subset construction procedures also admit synthesis of implementations. We also believe that the reusable LTS and CLTS libraries contributed by this work can serve as a starting point to mechanize other theoretical results related to CLTSs. Finally, we plan to incorporate other problems that are relevant to top-down verification and synthesis frameworks, such as subtyping and refinement.

References

- 1 Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. *IEEE Trans. Software Eng.*, 29(7):623–633, 2003. doi:10.1109/TSE.2003.1214326.
- 2 Rajeev Alur and Mihalis Yannakakis. Model checking of message sequence charts. In Jos C. M. Baeten and Sjouke Mauw, editors, *CONCUR ’99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24–27, 1999, Proceedings*, volume 1664 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 1999. doi:10.1007/3-540-48320-9_10.
- 3 Laura Bocchi, Romain Demangeon, and Nobuko Yoshida. A multiparty multi-session logic. In Catuscia Palamidessi and Mark Dermot Ryan, editors, *Trustworthy Global Computing - 7th International Symposium, TGC 2012, Newcastle upon Tyne, UK, September 7–8, 2012, Revised Selected Papers*, volume 8191 of *Lecture Notes in Computer Science*, pages 97–111. Springer, 2012. doi:10.1007/978-3-642-41157-1_7.
- 4 Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31–September 3, 2010. Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 2010. doi:10.1007/978-3-642-15375-4_12.
- 5 Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983. doi:10.1145/322374.322380.
- 6 Luís Caires and Jorge A. Pérez. Multiparty session types within a canonical binary theory, and beyond. In Elvira Albert and Ivan Lanese, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6–9, 2016, Proceedings*, volume 9688 of *Lecture Notes in Computer Science*, pages 74–95. Springer, 2016. doi:10.1007/978-3-319-39570-8_6.

- 685 **7** David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. Zooid: a DSL
686 for certified multiparty computation: from mechanised metatheory to certified multiparty
687 processes. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN*
688 *International Conference on Programming Language Design and Implementation, Virtual Event,*
689 *Canada, June 20-25, 2021*, pages 237–251. ACM, 2021. doi:10.1145/3453483.3454041.
- 690 **8** Tzu-Chun Chen. Lightening global types. *J. Log. Algebraic Methods Program.*, 84(5):708–729,
691 2015. URL: <https://doi.org/10.1016/j.jlamp.2015.06.003>, doi:10.1016/J.JLAMP.2015.
692 06.003.
- 693 **9** Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. *Theor.*
694 *Comput. Sci.*, 802:38–66, 2020. doi:10.1016/j.tcs.2019.07.005.
- 695 **10** Pierre-Malo Deniérou and Nobuko Yoshida. Multiparty session types meet communicating
696 automata. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European*
697 *Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on*
698 *Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012.*
699 *Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 194–213. Springer,
700 2012. doi:10.1007/978-3-642-28869-2_10.
- 701 **11** Pierre-Malo Deniérou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised
702 multiparty session types. *Log. Methods Comput. Sci.*, 8(4), 2012. doi:10.2168/LMCS-8(4:
703 6)2012.
- 704 **12** Burak Ekici and Nobuko Yoshida. Completeness of asynchronous session tree subtyping in Coq.
705 In *15th International Conference on Interactive Theorem Proving, ITP 2024, September 9-14,*
706 *2024, Tbilisi, Georgia*, volume 309 of *LIPICs*, pages 13:1–13:20. Schloss Dagstuhl - Leibniz-
707 Zentrum für Informatik, 2024. URL: <https://doi.org/10.4230/LIPICs.ITP.2024.13>, doi:
708 10.4230/LIPICs.ITP.2024.13.
- 709 **13** Alain Finkel and Étienne Lozes. Synchronizability of communicating finite state machines is
710 not decidable. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl,
711 editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP*
712 *2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPICs*, pages 122:1–122:14. Schloss
713 Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.ICALP.2017.122.
- 714 **14** Thomas Gazagnaire, Blaise Genest, Loïc Hélouët, P. S. Thiagarajan, and Shaofa Yang. Causal
715 message sequence charts. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR*
716 *2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal,*
717 *September 3-8, 2007, Proceedings*, volume 4703 of *Lecture Notes in Computer Science*, pages
718 166–180. Springer, 2007. doi:10.1007/978-3-540-74407-8_12.
- 719 **15** Blaise Genest and Anca Muscholl. Message sequence charts: A survey. In *Fifth International*
720 *Conference on Application of Concurrency to System Design (ACSD 2005), 6-9 June 2005, St.*
721 *Malo, France*, pages 2–4. IEEE Computer Society, 2005. doi:10.1109/ACSD.2005.25.
- 722 **16** Blaise Genest, Anca Muscholl, and Doron A. Peled. Message sequence charts. In Jörg Desel,
723 Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets,*
724 *Advances in Petri Nets [This tutorial volume originates from the 4th Advanced Course on Petri*
725 *Nets, ACPN 2003, held in Eichstätt, Germany in September 2003. In addition to lectures given*
726 *at ACPN 2003, additional chapters have been commissioned]*, volume 3098 of *Lecture Notes in*
727 *Computer Science*, pages 537–558. Springer, 2003. doi:10.1007/978-3-540-27755-2_15.
- 728 **17** Blaise Genest, Anca Muscholl, Helmut Seidl, and Marc Zeitoun. Infinite-state high-level
729 mscs: Model-checking and realizability. *J. Comput. Syst. Sci.*, 72(4):617–647, 2006. doi:
730 10.1016/j.jcss.2005.09.007.
- 731 **18** Lorenzo Gheri, Ivan Lanese, Neil Sayers, Emilio Tuosto, and Nobuko Yoshida. Design-by-
732 contract for flexible multiparty session protocols. In Karim Ali and Jan Vitek, editors,
733 *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022,*
734 *Berlin, Germany*, volume 222 of *LIPICs*, pages 8:1–8:28. Schloss Dagstuhl - Leibniz-Zentrum
735 für Informatik, 2022. URL: <https://doi.org/10.4230/LIPICs.ECOOP.2022.8>, doi:10.4230/
736 LIPICs.ECOOP.2022.8.

- 737 **19** Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, Alceste Scalas, and Nobuko Yoshida.
738 Precise subtyping for synchronous multiparty sessions. *J. Log. Algebraic Methods Program.*,
739 104:127–173, 2019. URL: <https://doi.org/10.1016/j.jlamp.2018.12.002>, doi:10.1016/J.
740 JLAMP.2018.12.002.
- 741 **20** Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi,
742 and Pascal Weisenburger. Multiparty languages: The choreographic and multitier cases (pearl).
743 In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented*
744 *Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume
745 194 of *LIPICs*, pages 22:1–22:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
746 doi:10.4230/LIPICs.ECOOP.2021.22.
- 747 **21** Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: session-type
748 based reasoning in separation logic. *Proc. ACM Program. Lang.*, 4(POPL):6:1–6:30, 2020.
749 doi:10.1145/3371074.
- 750 **22** Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris 2.0: Asynchronous
751 session-type based reasoning in separation logic. *Log. Methods Comput. Sci.*, 18(2), 2022.
752 URL: [https://doi.org/10.46298/lmcs-18\(2:16\)2022](https://doi.org/10.46298/lmcs-18(2:16)2022), doi:10.46298/LMCS-18(2:16)2022.
- 753 **23** Jonas Kastberg Hinrichsen, Jules Jacobs, and Robbert Krebbers. Multiris: Functional verifica-
754 tion of multiparty message passing in separation logic. 2024. URL: https://jihgfee.github.io/papers/multiris_manuscript.pdf.
755
- 756 **24** Jonas Kastberg Hinrichsen, Daniël Louwink, Robbert Krebbers, and Jesper Bengtson.
757 Machine-checked semantic session typing. In Catalin Hritcu and Andrei Popescu, ed-
758 itors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs*
759 *and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 178–198. ACM, 2021.
760 doi:10.1145/3437992.3439914.
- 761 **25** Andrew K. Hirsch and Deepak Garg. Pirouette: higher-order typed functional choreographies.
762 *Proc. ACM Program. Lang.*, 6(POPL):1–27, 2022. doi:10.1145/3498684.
- 763 **26** Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In
764 George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT*
765 *Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California,*
766 *USA, January 7-12, 2008*, pages 273–284. ACM, 2008. doi:10.1145/1328438.1328472.
- 767 **27** Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Connectivity graphs: a method for
768 proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang.*, 6(POPL):1–
769 33, 2022. doi:10.1145/3498662.
- 770 **28** Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. Dependent session protocols
771 in separation logic from first principles (functional pearl). *Proc. ACM Program. Lang.*,
772 7(ICFP):768–795, 2023. doi:10.1145/3607856.
- 773 **29** Ivan Lanese, Jorge A. Pérez, Davide Sangiorgi, and Alan Schmitt. On the expressiveness and
774 decidability of higher-order process calculi. In *Proceedings of the Twenty-Third Annual IEEE*
775 *Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*,
776 pages 145–155. IEEE Computer Society, 2008. doi:10.1109/LICS.2008.8.
- 777 **30** Elaine Li. Itp’25 artifact: Certified implementability of global multiparty protocols), 2025.
778 URL: <https://zenodo.org/records/15760397>, doi:10.5281/zenodo.15760396.
- 779 **31** Elaine Li, Felix Stutz, and Thomas Wies. Deciding subtyping for asynchronous multiparty
780 sessions. In Stephanie Weirich, editor, *Programming Languages and Systems - 33rd European*
781 *Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on*
782 *Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11,*
783 *2024, Proceedings, Part I*, volume 14576 of *Lecture Notes in Computer Science*, pages 176–205.
784 Springer, 2024. doi:10.1007/978-3-031-57262-3_8.
- 785 **32** Elaine Li, Felix Stutz, Thomas Wies, and Damien Zufferey. Complete multiparty session
786 type projection with automata. In Constantin Enea and Akash Lal, editors, *Computer Aided*
787 *Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023*,

- 788 *Proceedings, Part III*, volume 13966 of *Lecture Notes in Computer Science*, pages 350–373.
 789 Springer, 2023. doi:10.1007/978-3-031-37709-9_17.
- 790 33 Elaine Li, Felix Stutz, Thomas Wies, and Damien Zufferey. Characterizing implementability
 791 of global protocols with infinite states and data. *PACMPL*, 9(Object-oriented Programming,
 792 Systems, Languages, and Applications (OOPSLA)), 2025.
- 793 34 Elaine Li, Felix Stutz, Thomas Wies, and Damien Zufferey. Characterizing implementability
 794 of global protocols with infinite states and data, 2025. URL: <https://arxiv.org/abs/2411.05722v2>, arXiv:2411.05722v2.
- 795 35 Markus Lohrey. Realizability of high-level message sequence charts: closing the gaps. *Theor.*
 796 *Comput. Sci.*, 309(1-3):529–554, 2003. doi:10.1016/j.tcs.2003.08.002.
- 797 36 Rupak Majumdar, Madhavan Mukund, Felix Stutz, and Damien Zufferey. Generalising
 798 projection in asynchronous multiparty session types. In Serge Haddad and Daniele Varacca,
 799 editors, *32nd International Conference on Concurrency Theory, CONCUR 2021, August*
 800 *24-27, 2021, Virtual Conference*, volume 203 of *LIPIcs*, pages 35:1–35:24. Schloss Dagstuhl -
 801 Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CONCUR.2021.35.
- 802 37 Petar Maksimovic and Alan Schmitt. Hoco in coq. In Christian Urban and Xingyuan Zhang,
 803 editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing,*
 804 *China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*,
 805 pages 278–293. Springer, 2015. doi:10.1007/978-3-319-22102-1_19.
- 806 38 Sjouke Mauw and Michel A. Reniers. High-level message sequence charts. In Ana R. Cavalli and
 807 Amardeo Sarma, editors, *SDL '97 Time for Testing, SDL, MSC and Trends - 8th International*
 808 *SDL Forum, Evry, France, 23-29 September 1997, Proceedings*, pages 291–306. Elsevier, 1997.
- 809 39 Rémi Morin. Recognizable sets of message sequence charts. In Helmut Alt and Afonso Ferreira,
 810 editors, *STACS 2002, 19th Annual Symposium on Theoretical Aspects of Computer Science,*
 811 *Antibes - Juan les Pins, France, March 14-16, 2002, Proceedings*, volume 2285 of *Lecture Notes*
 812 *in Computer Science*, pages 523–534. Springer, 2002. doi:10.1007/3-540-45841-7_43.
- 813 40 Anca Muscholl and Doron A. Peled. Message sequence graphs and decision problems on
 814 mazurkiewicz traces. In Mirosław Kutylowski, Leszek Pacholski, and Tomasz Wierzbicki,
 815 editors, *Mathematical Foundations of Computer Science 1999, 24th International Symposium,*
 816 *MFCS'99, Szklarska Poreba, Poland, September 6-10, 1999, Proceedings*, volume 1672 of *Lecture*
 817 *Notes in Computer Science*, pages 81–91. Springer, 1999. doi:10.1007/3-540-48340-3_8.
- 818 41 Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed
 819 definitional interpreters for linear, session-typed languages. In Jasmin Blanchette and Catalin
 820 Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified*
 821 *Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 284–298.
 822 ACM, 2020. doi:10.1145/3372885.3373818.
- 823 42 Abhik Roychoudhury, Ankit Goel, and Bikram Sengupta. Symbolic message sequence charts.
 824 *ACM Trans. Softw. Eng. Methodol.*, 21(2):12:1–12:44, 2012. doi:10.1145/2089116.2089122.
- 825 43 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc.*
 826 *ACM Program. Lang.*, 3(POPL):30:1–30:29, 2019. doi:10.1145/3290343.
- 827 44 Gan Shen, Shun Kashiwa, and Lindsey Kuper. Haschor: Functional choreographic programming
 828 for all (functional pearl). *CoRR*, abs/2303.00924, 2023. URL: [https://doi.org/10.48550/](https://doi.org/10.48550/arXiv.2303.00924)
 829 [arXiv.2303.00924](https://doi.org/10.48550/arXiv.2303.00924), arXiv:2303.00924, doi:10.48550/ARXIV.2303.00924.
- 830 45 Felix Stutz. Asynchronous multiparty session type implementability is decidable - lessons
 831 learned from message sequence charts. In Karim Ali and Guido Salvaneschi, editors, *37th*
 832 *European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023,*
 833 *Seattle, Washington, United States*, volume 263 of *LIPIcs*, pages 32:1–32:31. Schloss Dagstuhl -
 834 Leibniz-Zentrum für Informatik, 2023. URL: [https://doi.org/10.4230/LIPIcs.ECOOP.2023.](https://doi.org/10.4230/LIPIcs.ECOOP.2023.32)
 835 [32](https://doi.org/10.4230/LIPIcs.ECOOP.2023.32), doi:10.4230/LIPIcs.ECOOP.2023.32.
- 836 46 Joseph Tassarotti, Ralf Jung, and Robert Harper. A higher-order logic for concurrent
 837 termination-preserving refinement. In Hongseok Yang, editor, *Programming Languages and*
 838 *Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the*
 839

- 840 *European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala,*
841 *Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*,
842 pages 909–936. Springer, 2017. doi:10.1007/978-3-662-54434-1_34.
- 843 47 Peter Thiemann. Intrinsically-typed mechanized semantics for session types. In Ekaterina
844 Komendantskaya, editor, *Proceedings of the 21st International Symposium on Principles and*
845 *Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, pages
846 19:1–19:15. ACM, 2019. doi:10.1145/3354166.3354184.
- 847 48 Dawit Legesse Tirore, Jesper Bengtson, and Marco Carbone. A sound and complete pro-
848 jection for global types. In Adam Naumowicz and René Thiemann, editor, *14th Interna-*
849 *tional Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023,*
850 *Białystok, Poland*, volume 268 of *LIPIcs*, pages 28:1–28:19. Schloss Dagstuhl - Leibniz-
851 Zentrum für Informatik, 2023. URL: <https://doi.org/10.4230/LIPIcs.ITP.2023.28>, doi:
852 10.4230/LIPIcs.ITP.2023.28.
- 853 49 Bernardo Toninho and Nobuko Yoshida. Certifying data in multiparty session types. *J. Log.*
854 *Algebraic Methods Program.*, 90:61–83, 2017. doi:10.1016/j.jlamp.2016.11.005.
- 855 50 Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida.
856 Statically verified refinements for multiparty protocols. *Proc. ACM Program. Lang.*,
857 4(OOPSLA):148:1–148:30, 2020. doi:10.1145/3428216.

858 Acknowledgements.

859 This work is supported by the National Science Foundation under the grant agreement
860 2304758.