The Concurrency Column

by

Marino Miculan and Nobuko Yoshida

Department of Mathematics, Computer Science and Physics
University of Udine
Via delle Scienze 206, Udine, Italy
marino.miculan@uniud.it

Department of Computer Science
University of Oxford
Wolfson Building, Parks Road, Oxford, UK
nobuko.yoshida@cs.ox.ac.uk

Welcome to this installment of *The Concurrency Column*. In this issue, we are pleased to present two outstanding contributions that significantly advance the formal verification and behavioral safety of message-passing concurrent systems through global protocol models.

The first contribution is by Elaine Li, summarizing the foundational theoretical and practical breakthroughs of her doctoral thesis, "Decision Problems for Global Protocol Specifications." As message-passing concurrency forms the bedrock of modern distributed systems, verifying communication protocols remains a critical challenge. High-level global protocol specifications (such as Message Sequence Charts, session types, and choreographies) offer a powerful top-down paradigm by describing interactions from a synchronous, bird's-eye view, ruling out major coordination errors by construction. However, implementing these specifications across asynchronous distributed networks introduces deep algorithmic complexities. In this article, Li addresses three central problems in global protocol verification: implementability, synthesis, and subtyping. Rather than treating these challenges in isolation, she provides a unifying semantic lens rooted in formal language and automata theory. This foundational approach yields precise, sound, and complete characterizations, highlighted by the formulation of "Coherence Conditions." No-

tably, these results are fully mechanized in the Rocq interactive theorem prover and implemented practically within the verification tool SPROUT.

The second contribution, “Typing Scalas-Yoshida Benchmarks” by Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de’ Liguoro, explores the practical expressive power of modern session type frameworks. The authors demonstrate how the challenging Scalas-Yoshida "Less Is More" benchmark protocols—including service-client authentication and recursive Map/Reduce—can be successfully represented and typed within the Simple Multiparty Sessions (SMPS) framework. By reconstructing suitable global types, they show that SMPS can seamlessly type protocols that are typically problematic or unprojectable in standard multiparty session type settings, all while firmly retaining critical behavioral guarantees like session fidelity and lock-freedom.

Together, these two papers provide an essential reference for researchers and practitioners designing robust, typesafe concurrent frameworks.

Decision Problems for Global Protocol Specifications

Elaine Li

Abstract

Concurrency is ubiquitous in modern computing, message passing is a major concurrency paradigm, and communication protocols are therefore a key target for formal verification. Global protocol specifications synchronously describe the message-passing behaviors of all protocol participants from a bird's-eye view, and thus rule out large classes of communication errors by construction. Global protocols are adopted in industry by the ITU standard and UML, and are widely studied in academia in the form of high-level message sequence charts, session types and choreographic programs. This paper summarizes theoretical and practical contributions from the author's Ph.D. thesis to three problems central to global protocol verification: implementability, synthesis, and subtyping. Implementability asks whether a protocol admits a distributed implementation, synthesis in turn computes one, and subtyping asks whether an admissible implementation can be substituted in whole or part to yield fewer behaviors. This paper situates these results in relation to one another, through the unifying semantic lens of formal language and automata theory.

1 Introduction

Concurrency is ubiquitous in modern computing, message passing is a major concurrency paradigm, and communication protocols are therefore a prime target for formal verification. Writing implementations for each protocol participant individually, such that their composition is free from communication errors and deadlocks, is challenging and error-prone. One salient methodology for verifying communication protocols centers around the abstraction of a *global protocol*. Global protocols specify the message-passing behaviors of all protocol participants synchronously, and from a bird's eye view, thus ruling out large classes of communication errors by construction. Global protocols are then used as a blueprint from which to synthesize correct-by-construction distributed implementations. Due to their attractive simplicity, global protocols are widely studied in academia, predominantly in the

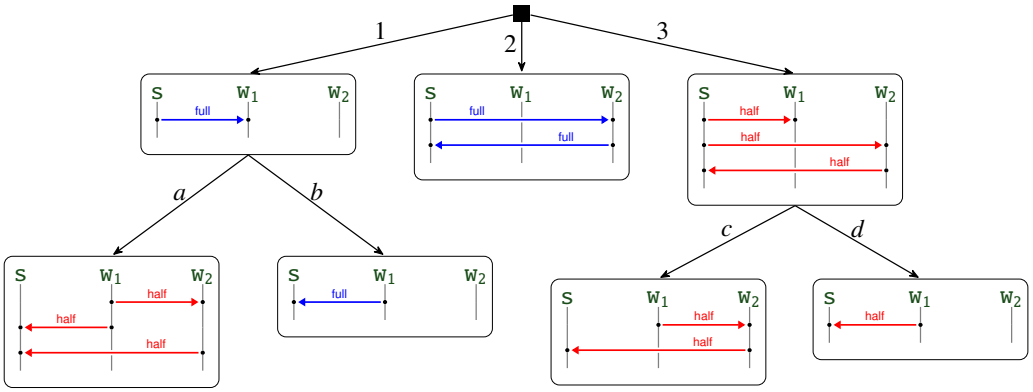

Figure 1: Task scheduling protocol.

form of high-level message sequence charts (HMSCs) [52], multiparty session types (MSTs) [29], and choreographic programs [13]. Message sequence charts represent global protocols using a graph-based visual formalism, and their theoretical aspects have been thoroughly investigated [23–25, 52, 64]. MSCs found early industry adoption by the ITU standard [33] in 1993, was subsequently incorporated into UML [60] in 2005, and is part of the Web Service Choreography Description Language [72]. MST frameworks represent global protocols as syntactic types, and provide an end-to-end workflow from local type projection to type checking of distributed processes. MST frameworks have been implemented in over a dozen programming languages, including Python [16, 55, 57], Java [30, 31], C [58], Go [6, 38], Scala [7], Rust [14, 36], OCaml [32] and F# [56]. Applications of MSTs include operating systems [19], high performance computing [28, 54, 59], cyber-physical systems [50, 51], and web services [74]. Choreographic programming frameworks represent global protocols as global programs, and have been implemented in Java [26], Haskell [66], Rust [34, 39] and applied to distributed architecture [61], cryptographic security protocols [22], and cyber-physical systems [12]. We refer the reader to [73] and [53] for a comprehensive survey of MST and choreography applicability respectively.

We use the example of a task scheduling protocol to illustrate the top-down verification methodology of global protocols, and introduce its key decision problems.

Global protocol specification. Fig. 1 depicts the global specification of the task scheduling protocol as an HMSC. In each HMSC vertex, vertical lines depict protocol participants, and horizontal arrows from the sender to the receiver depict synchronous message exchanges. HMSC edges depict global control flow constructs like recursion and non-deterministic choice. The task scheduling protocol involves three participants: a scheduler `s`, and two workers $\mathtt{w}_1$ and $\mathtt{w}_2$. The protocol

operates in a loop, but we omit back-edges to the initial vertex for readability. In each loop iteration, s chooses between assigning the entirety of a task to w_1 only, to w_2 only, or splitting the workload between the two workers. The first two cases correspond to branches 1 and 2, where s sends a full message to the respective worker. The third case corresponds to branch 3, where s sends a half message to both workers. Worker w_2 always completes its assigned task immediately and sends it back to s, by echoing the message it received from s. Worker w_1 may choose to behave like w_2 (branches *b* and *d*), or may choose to delegate some or all of its work to w_2 (branches *a* and *c*).

Implementability. The implementability problem was introduced for MSCs in [2], where it is referred to as *deadlock-free realizability*, or *safe realizability*. Implementability is also referred to as *projectability* in the MST literature. Implementability asks whether the global protocol, specified from the perspective of an omniscient coordinator, can be realized in a distributed setting, i.e. in the absence of such a coordinator, where participants can only gain information about the global protocol state by sending and receiving messages asynchronously. In particular, local participants should never deadlock, and should altogether behave consistently according to a single global run, executing send and receive actions in exactly the prescribed order. We refer to the latter property as *protocol fidelity*. Non-implementability arises when a participant's local information is insufficient for correctly determining its next action.

Synthesizing local implementations. If the protocol is implementable, the next step is to synthesize local implementations. Synthesis from specifications was first formulated by Church in 1957 [9] for logical circuits, and has been widely studied for different specification formalisms targeting different implementation models. Examples include distributed synthesis [20,35], reactive synthesis [62,63], syntax-guided synthesis [1], counterexample-guided inductive synthesis [67], and neurosymbolic program synthesis [8]. Synthesis is referred to as *projection* in the MST literature, and can be considered a special case in which the desired behavior is completely specified. Candidate local implementations for participants s, w_1 and w_2 are depicted in Figs. 2 to 4 respectively. Unlike the global protocol, which enjoys the illusion of synchrony, local implementations are asynchronous, i.e. sending and receiving are no longer specified atomically. Throughout the paper, we use $\mathtt{p} \triangleright \mathtt{q}!m$ to denote the send event where participant p sends *m* to q, and $\mathtt{q} \triangleleft \mathtt{p}?m$ to denote the receive event where q receives message *m* from p. In Figs. 2 to 4, the active participant is omitted from transition labels for clarity, e.g. in Fig. 2, transition label w_1!full denotes s sending full to w_1, and w_2?half denotes s receiving half from w_2.

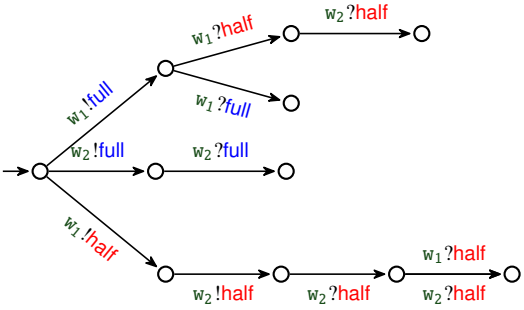

Figure 2: Local implementation for s

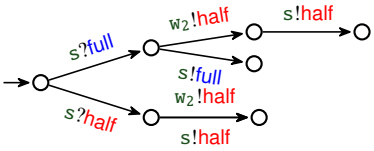

Figure 3: Local implementation for w_1

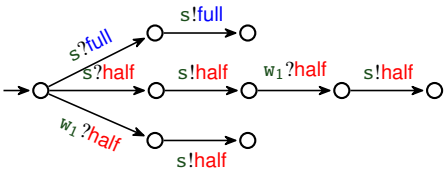

Figure 4: Local implementation for w_2

Subtyping and refinement. The task scheduling protocol offers w_1 multiple choice points at which to decide whether or not to delegate work to w_2. To reduce communication overhead and restrict non-determinism, we may wish to consider a *refinement* of the global protocol that removes some of the power of choice from w_1, for example by eliminating branch $1a$. The problems of *subtyping and refinement* ask whether the resulting global protocol remains implementable, and whether a given candidate implementation correctly implements it.

Implementability, synthesis and subtyping are very relevant and challenging problems, These have been studied using different techniques, most notably automata-based approaches within MSC and type-based systems within MST.

This paper presents sound and complete solutions to the aforementioned problems that improve prior work along the dimensions of expressivity, precision and optimality. This paper summarizes results from the author's Ph.D. thesis [40], parts of which have been published in [41–43, 45–47]. The summary aims to highlight the unifying semantic perspective adopted by [40] in addressing these problems: viewing global protocols and their local implementations as automata, and viewing their behavior as formal languages over an alphabet of asynchronous send and receive events. In particular, we show how the main contribution of [40], a characterization of implementability as a set of reachability and co-reachability 2-hyperproperties, called *Coherence Conditions*, can be used to derive the other obtained theoretical results, including matching complexity bounds for various finite protocol fragments, asymptotically optimal decision procedures for implementability, and sound and complete synthesis algorithms. The preciseness of the

characterization has been mechanized in the Rocq prover, and algorithms for deciding implementability have been implemented in a tool called Sprout. We further show that this characterization is naturally extensible along two orthogonal axes: to protocols with infinite states and data, and to alternative models of asynchrony beyond the commonly assumed peer-to-peer FIFO.

The remainder of the paper is structured as follows. In §2, we present preliminaries. In §3, we present our unifying representation of global protocols, called *Global Communicating Labeled Transition Systems* (GCLTS). We then present the set of Coherence Conditions (*CC*), sketch its equivalence to GCLTS implementability, and use it to derive synthesis and complexity results. In §4, we extend *CC* to the infinite-state setting, and in §5, we extend *CC* to the network-parametric setting. In §6, we briefly discuss aspects of *CC*'s mechanization and implementation.

2 Preliminaries

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 $\mathrm{pref}(u)$. 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 possibly infinite set of participants and $\mathcal{V}$ be a possibly infinite data domain. We define the set of *synchronous events* $\Gamma_{sync} := \{\mathtt{p} \to \mathtt{q} : m \mid \mathtt{p}, \mathtt{q} \in \mathcal{P} \text{ and } m \in \mathcal{V}\}$ where $\mathtt{p} \to \mathtt{q} : m$ denotes a message exchange of m from sender $\mathtt{p}$ to receiver $\mathtt{q}$. For a participant $\mathtt{p} \in \mathcal{P}$, we define the alphabet $\Gamma_{\mathtt{p}} := \{\mathtt{p} \to \mathtt{q} : m \mid \mathtt{q} \in \mathcal{P},\ m \in \mathcal{V}\} \cup \{\mathtt{q} \to \mathtt{p} : m \mid \mathtt{q} \in \mathcal{P},\ m \in \mathcal{V}\}$, denoting the set of synchronous events in which $\mathtt{p}$ is either the sender or receiver in the message exchange. A synchronous event is split into a send and receive *asynchronous event* for the sender and receiver respectively. For a participant $\mathtt{p} \in \mathcal{P}$, we define the alphabet $\Sigma_{\mathtt{p},!} := \{\mathtt{p} \triangleright \mathtt{q}!m \mid \mathtt{q} \in \mathcal{P},\ m \in \mathcal{V}\}$ of *send* events and the alphabet $\Sigma_{\mathtt{p},?} := \{\mathtt{p} \triangleleft \mathtt{q}?m \mid \mathtt{q} \in \mathcal{P},\ m \in \mathcal{V}\}$ of *receive* events. The asynchronous event $\mathtt{p} \triangleright \mathtt{q}!m$ denotes participant $\mathtt{p}$ sending a message m to $\mathtt{q}$, and $\mathtt{p} \triangleleft \mathtt{q}?m$ denotes participant $\mathtt{p}$ receiving a message m from $\mathtt{q}$. We write $\Sigma_{\mathtt{p}} = \Sigma_{\mathtt{p},!} \cup \Sigma_{\mathtt{p},?}$, $\Sigma_! = \bigcup_{\mathtt{p} \in \mathcal{P}} \Sigma_{\mathtt{p},!}$, and $\Sigma_? = \bigcup_{\mathtt{p} \in \mathcal{P}} \Sigma_{\mathtt{p},?}$. Finally, the set of *asynchronous* events is $\Sigma_{async} = \Sigma_! \cup \Sigma_?$.

Projections. We map synchronous words to asynchronous words using a homomorphism $\mathtt{split}$, defined as $\mathtt{split}(\mathtt{p} \to \mathtt{q} : m) := \mathtt{p} \triangleright \mathtt{q}!m.\, \mathtt{q} \triangleleft \mathtt{p}?m$. Because $\mathtt{split}$ is injective, there exists a unique inverse, which we denote $\mathtt{split}^{-1}$. We say that $\mathtt{p}$

is *active* in x if $x \in \Gamma_{sync}$ and $x \in \Gamma_{\mathtt{p}}$, or if $x \in \Sigma_{async}$ and $x \in \Sigma_{\mathtt{p}}$. For each participant $\mathtt{p} \in \mathcal{P}$, we define a homomorphism $\Downarrow_{\Gamma_{\mathtt{p}}}$, where $x\Downarrow_{\Gamma_{\mathtt{p}}} = x$ if $x \in \Gamma_{\mathtt{p}}$ and ε otherwise, and a homomorphism $\Downarrow_{\Sigma_{\mathtt{p}}}$, where $x\Downarrow_{\Sigma_{\mathtt{p}}} = x$ if $x \in \Sigma_{\mathtt{p}}$ and ε otherwise. We define a class of projections based on pattern-matching of alphabet symbols, denoted $\Downarrow_{\_}$. The result of the projection is determined by the unspecified parts of the pattern. For example, $\Downarrow_{\mathtt{p}\triangleright\_!\_}$ projects the event $\mathtt{p} \triangleright \mathtt{q}!m$ onto $(\mathtt{q}, m)$, and non-send events and send events that do not have $\mathtt{p}$ as the sender onto ε. The function $\Downarrow_{\mathtt{q}\triangleleft\mathtt{p}?\_}$ projects receive events of $\mathtt{p}$ from $\mathtt{q}$ of any message value onto the message value, and all other events to ε.

Our starting point for specifying global protocols 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 *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. An LTS is *deterministic* if for every $s \xrightarrow{x_1} s_1, s \xrightarrow{x_2} s_2 \in T$, $x_1 = x_2$ implies $s_1 = s_2$. 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).

Our distributed implementation model is based on communicating state machines (CSMs) [5]. CSMs consist of a collection of deterministic finite state machines, one for each participant, that communicate via pairwise FIFO channels. CLTS generalize CSMs by lifting the restriction that the number of participants and the state space of each local state machine must be finite.

Communicating LTS. $\mathcal{T} = \{T_{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}}$ is a *communicating labeled transition system* (CLTS) over $\mathcal{P}$ and $\mathcal{V}$ if $T_{\mathtt{p}}$ is a deterministic LTS over $\Sigma_{\mathtt{p}}$ for every $\mathtt{p} \in \mathcal{P}$, denoted by $(Q_{\mathtt{p}}, \Sigma_{\mathtt{p}}, \delta_{\mathtt{p}}, q_{0,\mathtt{p}}, F_{\mathtt{p}})$. Let $\prod_{\mathtt{p}\in\mathcal{P}} Q_{\mathtt{p}}$ denote the set of global states and $\mathsf{Chan} = \{(\mathtt{p}, \mathtt{q}) \mid \mathtt{p}, \mathtt{q} \in \mathcal{P}, \mathtt{p} \neq \mathtt{q}\}$ 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 $\vec{s}_{\mathtt{p}}$ to denote the state of $\mathtt{p}$ in $\vec{s}$. The CLTS transition relation, denoted $\rightarrow$, is defined as follows.

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

- $(\vec{s}, \xi) \xrightarrow{\mathtt{q} \triangleleft \mathtt{p}?m} (\vec{s}', \xi')$ if $(\vec{s}_{\mathtt{q}}, \mathtt{q} \triangleleft \mathtt{p}?m, \vec{s}'_{\mathtt{q}}) \in \delta_{\mathtt{q}}$, $\vec{s}_{\mathtt{r}} = \vec{s}'_{\mathtt{r}}$ for every participant $\mathtt{r} \neq \mathtt{q}$, $\xi(\mathtt{p}, \mathtt{q}) = m \cdot \xi'(\mathtt{p}, \mathtt{q})$ 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 $q_{0,\mathtt{p}}$ of $A_{\mathtt{p}}$, and ξ_0 maps each channel to ε. A configuration $(\vec{s}, \xi)$ is *final* iff $\vec{s}_{\mathtt{p}}$ is final for every $\mathtt{p}$ 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. Equivalently, a CLTS is deadlock-free if every trace has a maximal extension.

Note that CLTS fixes a peer-to-peer communication topology with FIFO queues as its message channels. We refer to the communication topology and channel data structure as the *network architecture*, denoted $\mathbb{A}$. We defer the formal definition of $\mathbb{A}$ to §5, where we also generalize the definition of CLTS to be parametric in the choice of $\mathbb{A}$; for now fix $\mathbb{A}$ to be peer-to-peer communication over FIFO channels.

Channel Compliance. Before we define the semantics of global protocols, it is useful to disentangle the behavior of a given network architecture $\mathbb{A}$ from that of any particular CLTS $\mathcal{T}$. We extend our semantic perspective to network architectures, and introduce the notion of *channel compliance*, which describes the set of all words consistent with $\mathbb{A}$. We define channel compliance in terms of a *universal* CLTS whose local implementations accept all traces. Thus, the only restrictions on traces of a universal CLTS are imposed by the network architecture $\mathbb{A}$. Formally, let $\mathcal{U}_{\mathbb{A}}$ be the *universal CLTS* for $\mathbb{A}$, with $\mathcal{U}_{\mathbb{A}} = \{U_{\mathtt{p}}\}_{\mathtt{p} \in \mathcal{P}}$ such that $\mathcal{L}(U_{\mathtt{p}}) = \Sigma^*_{\mathtt{p}}$ for every $\mathtt{p} \in \mathcal{P}$. We say that $w \in \Sigma^*_{async}$ is $\mathbb{A}$-channel-compliant if w is a trace of $\mathcal{U}_{\mathbb{A}}$. If $\mathbb{A}$ is understood, we simply say w is channel-compliant. Moreover, if in fact $w \in \mathcal{L}(\mathcal{U}_{\mathbb{A}})$ holds, then we call w *channel-matched*. Intuitively, if in any given CLTS for $\mathbb{A}$ one can execute a channel-matched word, then in the reached configuration all buffers will be empty. We use $\mathcal{L}(\mathbb{A}) \subseteq \Sigma^*_{async}$ to denote all channel-compliant words.

Global Protocol Semantics. We next define the asynchronous semantics of global protocol $\mathcal{S}$, denoted $C^{\sim}(\mathcal{S}) \subseteq \Sigma^{\infty}$. The starting point for the semantics $C^{\sim}(\mathcal{S})$ is the synchronous language $\mathcal{L}(\mathcal{S})$. A word in $\mathcal{L}(\mathcal{S})$ specifies the coordination of events across all protocol participants, in addition to a total order of events per participant. From $\mathcal{L}(\mathcal{S})$ we obtain a set of 1-synchronous asynchronous words through `split`, which simply splits each atomic message exchange into its send and receive counterparts, denoted $\mathtt{split}(\mathcal{L}(\mathcal{S}))$. We want to include all asynchronous words that are equal to these 1-synchronous words under local projection and the given network architecture $\mathbb{A}$.

We handle the finite and infinite words separately to define the global protocol semantics as the union of its finite and infinite semantics:

$$C^{\sim}(\mathcal{S}) = C^{\sim}_{\text{fin}}(\mathcal{S}) \cup C^{\sim}_{\text{inf}}(\mathcal{S})$$

The finite semantics is obtained by following the above recipe, but restricting $\texttt{split}(\mathcal{L}(\mathcal{S}))$ to finite words:

$$C^{\sim}_{\text{fin}}(\mathcal{S}) = [\Sigma^{*}_{async} \cap \texttt{split}(\mathcal{L}(\mathcal{S}))]_{\equiv_{\mathcal{P}}} \cap \mathcal{L}(\mathbb{A}) \ .$$

The infinite semantics are those words whose prefixes are extensible to some word in $\mathcal{L}(\mathcal{S})$ modulo equality under local projection and the network semantics:

$$C^{\sim}_{\text{inf}}(\mathcal{S}) = \{w \in \Sigma^{\infty}_{async} \mid \forall u \leq w.\ u \in \text{pref}([\texttt{split}(\mathcal{L}(\mathcal{S}))]_{\equiv_{\mathcal{P}}} \cap \mathcal{L}(\mathbb{A}))\} \ .$$

Finally, we define protocol implementability.

Protocol Implementability. A protocol $\mathcal{S}$ is *implementable* if there exists a CLTS $\{T_{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}}$ satisfying the following two properties:

(i) *protocol fidelity*: $\mathcal{L}(\{T_{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}}) = \mathcal{L}(\mathcal{S})$, and

(ii) *deadlock freedom*: $\{T_{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}}$ is deadlock-free.

We say that such a $\{T_{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}}$ implements $\mathcal{S}$.

3 Implementability

3.1 Motivation

We first analyze the implementability of the task scheduling protocol in Fig. 1 by a CLTS with peer-to-peer FIFO channels. Consider the two executions u_1 and u_2.

$u_1 = \mathtt{s} \triangleright \mathtt{w}_1!\mathsf{full} \cdot \mathtt{w}_1 \triangleright \mathtt{w}_2!\mathsf{half}$ $\qquad$ $u_2 = \mathtt{s} \triangleright \mathtt{w}_1!\mathsf{half} \cdot \mathtt{w}_1 \triangleleft \mathtt{s}?\mathsf{half} \cdot \mathtt{w}_1 \triangleright \mathtt{w}_2!\mathsf{half}$

In u_1, participants $\mathtt{w}_1$ and $\mathtt{s}$ decide to follow branch $1a$, whereas in u_2, they decide to follow branch $3c$. However, from the perspective of $\mathtt{w}_2$, who has observed no events in the protocol thus far, the two executions are indistinguishable: both lead to the same local state q, in which only half from $\mathtt{w}_1$ is available in the channel between $\mathtt{w}_1$ and $\mathtt{w}_2$ (the half from $\mathtt{s}$ is delayed in u_2). Thus, in a distributed setting, $\mathtt{w}_1$ cannot tell the two executions apart. In order to correctly follow the protocol, $\mathtt{w}_2$'s next action in u_1 should be to send half to $\mathtt{s}$, whereas in u_2, $\mathtt{w}_2$ should first wait for the arrival of half from $\mathtt{s}$. If $\mathtt{w}_2$ were to wait in state q, this would lead u_1 to deadlock, and if $\mathtt{w}_2$ were to send half to server, this would lead u_2 to violate protocol fidelity. Thus, the protocol in Fig. 1 is not implementable.

A possible repair is to replace the message value half sent from $\mathtt{w}_1$ to $\mathtt{w}_2$ on branch $1a$ with delegate. Now, $\mathtt{w}_2$ can tell the two branches apart: it can wait until either half is available in its channel from $\mathtt{s}$, indicating that the other participants are following branch 3, or delegate is available in the channel from $\mathtt{w}_1$, indicating that the other participants are following branch $1a$. Since the two cases are exclusive, $\mathtt{w}_2$ can make its decision as soon as it observes one of the two. This change renders the protocol implementable.

3.2 Global Communicating Labeled Transition Systems

Unfortunately, the implementability problem for global protocols is undecidable in general [48]. To restore decidability, we impose three conditions on the class of global protocols as LTS we consider that are inspired by the syntactic structure of global multiparty session types.

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

(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 $\mathtt{p} \in \mathcal{P}$ who is the sender for both, i.e. $\mathtt{split}(l_i) \in \Sigma_{\mathtt{p},!}$ 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 only required to ensure that the *finite* language of an implementation for global protocol $\mathcal{S}$ matches the *finite* semantics of $\mathcal{S}$. The condition can be waived if we define the semantics of our implementation model in terms of traces. Condition (2) generalizes classical multiparty session types fragments, which require not only a dedicated sender but also a dedicated receiver, a condition we refer to as *directed choice*. In contrast, *mixed choice* lifts all restrictions on choice, and amounts to only requiring determinism. [48] showed that realizability is undecidable for high level message sequence charts satisfying determinism and Condition (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 this paper, we refer to a GCLTS simply as a *protocol*.

3.3 Coherence Conditions

The non-implementability in our running example can be attributed to insufficient local information about protocol control flow. Existing MST works soundly detect insufficient local information through conservative, type-based projection algorithms [11, 29, 65, 68], which are partial functions that map global types to collections of local types, and are only defined for implementable global types. In other words, projection algorithms solve implementability and synthesis altogether in one shot. [42] presents an automata-based projection algorithm that is both sound and complete, but still requires synthesizing local implementations upfront to check implementability, which is in PSPACE.

Our approach instead solves implementability by analyzing the global protocol directly, without synthesizing a candidate implementation upfront. This is especially significant in the infinite case, discussed in §4, when synthesizing a candidate implementation is itself challenging and not always possible. Our key insight is that non-implementability can be blamed solely on the existence of certain pairs of states in the global protocol. Specifically, non-implementability arises when two states are *locally indistinguishable* to a participant — that is, they are both reachable via the same sequence of that participant's local events — yet require that participant to behave differently. This observation naturally gives rise to a characterization of implementability as 2-hyperproperties with a non-interference flavor. We develop this characterization, called the Coherence Conditions, in the remainder of the section.

Let $\mathcal{S} = (S, \Gamma_{sync}, T, s_0, F)$ be a protocol and let $\mathtt{p} \in \mathcal{P}$ be a participant. We begin by formalizing a participant's local information about the protocol using two variations on the standard notion of reachability. 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* $\mathtt{p}$ *on* $u \in \Gamma_{\mathtt{p}}^*$ when there exists $w \in \Gamma_{sync}^*$ such that $s_0 \xrightarrow{w}{}^* s \in T$ and $w{\Downarrow_{\Gamma_{\mathtt{p}}}} = u$, which we denote $s_0 \overset{u}{\underset{\mathtt{p}}{\Rightarrow}}{}^* s$.

Simultaneous Reachability. We say that $s_1, s_2 \in S$ are *simultaneously reachable* for participant $\mathtt{p}$ on $u \in \Gamma_{\mathtt{p}}^*$, denoted $s_0 \overset{u}{\underset{\mathtt{p}}{\Rightarrow}}{}^* s_1, s_2$, 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 $w_1{\Downarrow_{\Gamma_{\mathtt{p}}}} = w_2{\Downarrow_{\Gamma_{\mathtt{p}}}} = u$.

Simultaneous reachability captures the notion of *locally indistinguishable* states: to a participant, two states are locally indistinguishable if they are simultaneously reachable.

Next, we analyze the possible asynchronous behavior of a participant when it is in two locally indistinguishable states, starting from send events. Since send events are always enabled, and are moreover committal to a particular protocol run, we require 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. This condition is captured by Send Coherence, defined below.

Definition 1 (Send Coherence)**.** $\mathcal{S}$ satisfies Send Coherence (SC) when for every $s_1 \xrightarrow{\mathtt{p}\to\mathtt{q}:m} s_2 \in T, s_1' \in S$:

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

Next, we consider receive events. Unlike send events, receive events are not always enabled. The availability of messages from other participants encodes information about the global protocol state, and receptions correspondingly update the receiver's local view. The information-carrying role that receptions serve, coupled with the fact that in each protocol run, a participant's events are totally ordered, gives rise to Receive Coherence, which requires that receptions must uniquely distinguish two locally indistinguishable states for the receiver.

Definition 2 (Receive Coherence)**.** $\mathcal{S}$ satisfies Receive Coherence (RC) when for every $s_1 \xrightarrow{\mathtt{p}\to\mathtt{q}:m} s_2, s_1' \xrightarrow{\mathtt{r}\to\mathtt{q}:m} s_2' \in T$ with $\mathtt{r} \neq \mathtt{p}$:

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

Finally, we require that participants cannot equivocate between sending and receiving messages in two locally indistinguishable states. We call this condition No Mixed Choice.

Definition 3 (No Mixed Choice)**.** A protocol $\mathcal{S} = (S, \Gamma_{sync}, T, s_0, F)$ satisfies No Mixed Choice (NMC) when for every $s_1 \xrightarrow{\mathtt{p}\to\mathtt{q}:m} s_2, s_1' \xrightarrow{\mathtt{r}\to\mathtt{p}:m} s_2' \in T$:

$$(\exists u \in \Gamma_{\mathtt{p}}^*.\ s_0 \overset{u}{\underset{\mathtt{p}}{\Rightarrow}}{}^* s_1, s_1') \implies \bot \ .$$

Our semantic characterization of protocol implementability is the conjunction of the above three conditions.

Definition 4 (Coherence Conditions)**.** A protocol satisfies Coherence Conditions (CC) when it satisfies Send Coherence, Receive Coherence and No Mixed Choice.

We show that *CC* is equivalent to implementability, i.e. it soundly and completely characterizes implementable protocols.

Theorem 5 (Preciseness of Coherence Conditions)**.** *Let $\mathcal{S}$ be a protocol. Then, $\mathcal{S}$ is implementable if and only if it satisfies CC.*

The proof of Theorem 5 is fully mechanized in Rocq. We sketch the proof, in particular that of soundness, which has immediate consequences for synthesis.

Soundness requires showing that any protocol satisfying *CC* is implementable. We choose the *canonical implementation* as the existential witness: a CLTS $\{T_{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}}$ is a canonical implementation for $\mathcal{S}$ if for every $\mathtt{p} \in \mathcal{P}$, $T_{\mathtt{p}}$ satisfies $\mathcal{L}(T_{\mathtt{p}}) = \mathcal{L}(\mathcal{S})\Downarrow_{\Sigma_{\mathtt{p}}}$ and $\mathrm{pref}(\mathcal{L}(T_{\mathtt{p}})) = \mathrm{pref}(\mathcal{L}(\mathcal{S})\Downarrow_{\Sigma_{\mathtt{p}}})$. The canonical implementation is an obvious choice because it contains exactly the behaviors prescribed by the global protocol: no more, and no less. Thus, it is easy to show that the canonical implementation exhibits at least the global protocol's behaviors. On the contrary, it is challenging to show that the canonical implementation does not introduce new, unspecified behaviors. The proof amounts to showing that the invariant of possible run set non-emptiness is inductive thanks to the Coherence Conditions.

Completeness is proved constructively by modus tollens: assuming the negation of each Coherence Condition in turn, we construct a concrete trace that any implementation must admit, but which is not a prefix of the protocol language, thereby violating either protocol fidelity or deadlock freedom. The modularity of the Coherence Conditions means that implementability violations can be localized to individual transitions for a given participant, a fact that Sprout exploits for actionable error reporting.

3.4 Synthesis

The role that the canonical implementation plays in the proof of soundness immediately yields the following corollary.

Corollary 6 (Canonical implementation is all you need)**.** *A protocol $\mathcal{S}$ is implementable if and only if its canonical implementation implements it.*

Furthermore, the canonical implementation's definition immediately suggests a direct synthesis algorithm that is sound and complete thanks to the above: first, project the global protocol onto each participant's local alphabet, then determinize the result. For finite global protocols, this amounts to a modified subset construction, and, recalling that space complexity disregards output tape usage, yields the following complexity result for the synthesis problem:

Theorem 7 (Synthesis of finite global protocols is in PSPACE)**.** *For finite protocols, the canonical implementation for each participant can be computed in PSPACE.*

The exponential state space blowup incurred by subset construction cannot be avoided when used for synthesizing canonical implementations. The following family of implementable global protocols exhibits the exponential blowup. We represent the global protocols as global types $\mathbf{G}_n$ for $n \in \mathbb{N}$ such that $\mathbf{G}_n$ has size linear in n. In global types, synchronous message exchanges $\mathtt{p} \to \mathtt{q} : m$ appear along with + denoting choice, μt binding a recursion variable t, and 0 denoting termination. The construction of the $\mathbf{G}_n$'s builds on the regular expression $(a^*(ab^*)^n a)^*$, which can only be recognized by a deterministic finite state machine that grows exponentially with n [17, Theorem 11].

Definition 8 (Global type family $\mathbf{G}_n$ with exponential canonical implementation)**.** $\mathbf{G}_n \coloneqq G(G_n)$, with G_i and G(-) are defined as:

$$G_0 \coloneqq \mathtt{p} \to \mathtt{q} : a.\, t_1 \quad \text{and} \quad G_i \coloneqq \mathtt{p} \to \mathtt{q} : a.\, \mu t_{3,i}.\, + \begin{cases} \mathtt{p} \to \mathtt{r} : m_3.\, \mathtt{p} \to \mathtt{q} : b.\, t_{3,i} \\ \mathtt{p} \to \mathtt{r} : n_3.\, G_{i-1} \end{cases} \quad \text{for } i > 0$$

$$G(G') \coloneqq \mu t_1.\, + \begin{cases} \mathtt{p} \to \mathtt{r} : m_1.\, \mu t_2.\, + \begin{cases} \mathtt{p} \to \mathtt{r} : m_2.\, \mathtt{p} \to \mathtt{q} : a.\, t_2 \\ \mathtt{p} \to \mathtt{r} : n_2.\, G' \end{cases} \\ \mathtt{p} \to \mathtt{r} : n_1.\, 0 \end{cases} .$$

Each G_i for $i > 0$ generates (ab^*), G_0 adds the last a, and the outermost Kleene star and the first a^* are defined using the scaffolding provided by G(-). Participant $\mathtt{p}$'s choice to send either m_3 or n_3 to $\mathtt{r}$ respectively encodes the choice to continue iterating b's or to stop in b^*; participant $\mathtt{q}$ however, is not involved in this exchange and thus $\mathtt{q}$'s local language is isomorphic to $(ab^*)^i a$. As $\mathbf{G}_n$ is implementable, the subset construction for each participant implements it by Corollary 6. By the definition of canonicity, we have $\mathcal{L}(\mathbf{G}_n)\Downarrow_{\Sigma_\mathtt{q}}$ for participant $\mathtt{q}$ over alphabet $\Sigma_\mathtt{q}$. Because $\mathcal{L}(\mathbf{G}_n)\Downarrow_{\Sigma_\mathtt{q}}$ can only be recognized by a deterministic finite state machine with size exponential in n, the corresponding local language preserving implementation also has size exponential in n.

Like implementability, synthesis is undecidable for the general class of protocols. However, for many expressive fragments of protocols that still feature infinite data, e.g. symbolic finite automata [15, 66] and certain classes of timed and register automata [4, 10], off-the-shelf determinization algorithms [3, 70, 71] can be used directly to compute canonical implementations.

3.5 Complexity

CC directly suggests the following algorithm for deciding implementability of protocols with finitely many states and transitions.

We defer the formal definition of avail to the next section. Using *CC*, we can also establish that implementability of finite protocols is co-NP-complete.

Algorithm 1 Checking *CC* for finite protocols

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

Theorem 9 (Complexity of implementability for finite protocols)**.** *Implementability of finite protocols is co-NP-complete.*

Proof sketch. 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 $\mathtt{p}$ and a pair of states s_1, s_2 satisfying the respective preconditions, and verifying simultaneous reachability of s_1 and s_2 for $\mathtt{p}$. For Receive Coherence, $\mathsf{avail}_{\mathtt{p},\mathtt{q},\{\mathtt{q}\}}(m, s_2)$ can be checked in NP by guessing a simple path in $\mathcal{S}$ from s_2 to a state with an outgoing transition $\mathtt{p} \to \mathtt{q} : m$, and evaluating the availability predicate along that path in polynomial time. We can restrict ourselves to simple paths because the blocked set $\mathcal{B}$ monotonically increases along a path in $\mathcal{S}$, and avail is antitone in $\mathcal{B}$.

For co-NP-hardness, we reduce 3-SAT to non-implementability. Given a 3-SAT instance $\varphi = C_1 \wedge \ldots \wedge C_k$ with variables $x_1, \ldots, x_n$ and literals L_{ij}, denoting the jth literal of clause C_i, with $1 \leq i \leq k$ and $1 \leq j \leq 3$, we construct a global protocol $\mathcal{S}_\varphi$ such that ϕ is unsatisfiable iff $\mathcal{S}_\varphi$ is implementable.

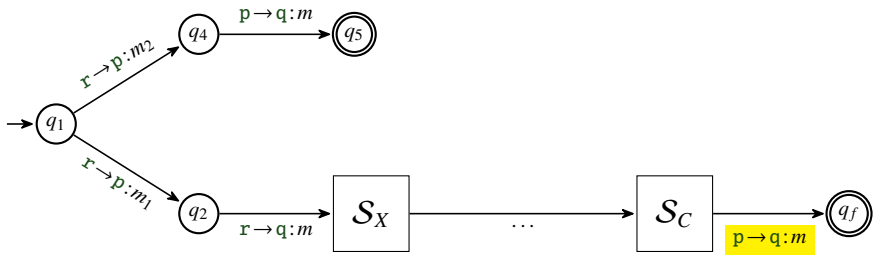

Figure 5: 3-SAT reduction to non-implementability.

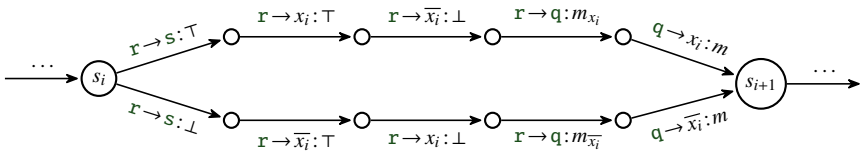

Figure 6: Variable assignment gadget $\mathcal{S}_X$.

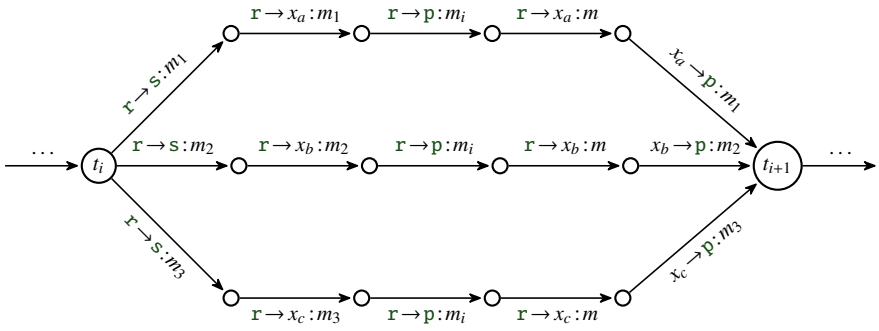

Figure 7: Clause selection gadget $\mathcal{S}_C$.

The construction (summarized pictorially in Fig. 5) relies on two gadgets: $\mathcal{S}_X$, a gadget that encodes a variable assignment to variables $x_1, \ldots, x_n$ (Fig. 6), and $\mathcal{S}_C$, a gadget that encodes literal selection for clauses $C_1, \ldots, C_k$ (Fig. 7). The highlighted message in Fig. 5 is available for participant q in q_2 if and only if φ is satisfiable. Consequently, protocol $\mathcal{S}_\varphi$ is non-implementable if and only if the highlighted message is available for participant q in q_2, if and only if φ is satisfiable. □

Observe that $\mathcal{S}_\varphi$, $\mathcal{S}_X$ and $\mathcal{S}_C$ all satisfy directed choice. Thus, the co-NP hardness lower bound holds even for the syntactically defined fragment of multiparty session types with directed choice [29], and we have the following corollary.

Corollary 10. *Implementability of global multiparty session types is co-NP-complete.*

4 Extension to Infinite Protocols

In this section, we generalize *CC* to infinite-state protocols. We first illustrate how our task delegation protocol can be made infinite-state, then introduce our formalism for specifying infinite-state protocols, called *symbolic protocols*, and finally show how to generalize *CC* to symbolic protocols.

4.1 Motivation

Our repaired task delegation protocol features finitely many message payloads: half, full and done. To model real-world verification targets, we desire for our global protocol specifications to be as expressive as possible. For example, we may wish to express that in branch 3, s can choose to split the workload between w_1 and w_2 into two parts l_1 and l_2, which are two reals that sum up to 1. Thus, there are infinitely many choices of l_1 and l_2, and the global protocol, in addition to specifying control flow, now also specifies data dependencies expressed as numeric constraints.

Data dependencies make the implementability problem significantly more challenging to analyze because it can influence control flow without relying on branching choice, as illustrated in the following pair of examples $\mathbb{S}_1$ (using ⓐ) and $\mathbb{S}_1'$ (using ⓑ) in Fig. 8. In both versions, participant p chooses a branch, even or odd, 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. If r cannot distinguish the two branches, then it cannot tell which value was sent first, and may subtract the values in the wrong order. This is the case in ⓐ, in which r receives a value y from p that satisfies constraint $y > x$, but does not reveal any information about p's next choice. In ⓑ, however, with the constraint on y changed from $y > x$ to $y = x + 2$, the value y now indirectly encodes p's next choice of even or odd through its parity, which in turn informs r of the order in which it should receive messages from p and q. In other words, the incorporation of dependent refinements also enables a new method of protocol repair: one that does not change the communication structure of the original protocol.

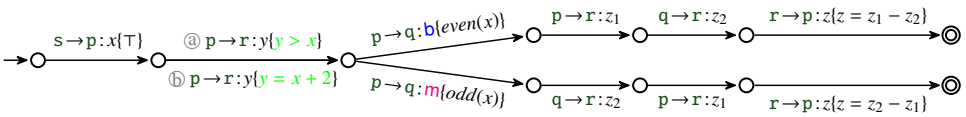

Figure 8: Two protocols: $\mathbb{S}_1$ using ⓐ with receive order violation $\mathbb{S}_1'$ using ⓑ without receive order violation.

We address the additional complexity introduced by data refinements with a semantic separation of concerns. First, we make the key observation that the Coherence Conditions remain sound and complete with respect to implementability even for protocols with infinitely many states and data. Unlike in the finite case, *CC* cannot be checked directly, i.e. by iterating over all states and transitions. Thus, the only remaining task is to reformulate the Coherence Conditions in a symbolic, semantics-preserving way that is effectively checkable for some finite representation of infinite protocols. We introduce this representation, called *symbolic protocols*, below.

In other words, we want to define symbolic conditions that are valid on the symbolic protocol if and only if its concrete protocol is implementable. Specifically, we provide a sound and complete reduction to the first-order fixpoint logic μCLP [69]. Validity of these formulas can then be discharged using off-the-shelf tools, which we demonstrate in the Sprout tool. 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. The expressive power of μCLP is needed because Send Coherence reasons about both reachability and coreachability, which naturally translate to least and greatest fixpoints over the protocol's transition relation. The reduction thus constitutes an algorithm that is complete relative to an assumed oracle for solving μCLP satisfiability.

4.2 Symbolic Protocols

Our definition of symbolic protocols assumes a fixed but unspecified first-order background theory of message values (e.g. linear real 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 and 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 11. 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, $F \subseteq S$ is a set of final 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' with the sent value x,
- $s_0 \in S$ is the initial control state,
- $\rho_0 \colon R \to \mathcal{V}$ is the initial register assignment.

In our definition, the transition constraint φ altogether specifies constraints on the message value and register updates. To this end, for each register variable $r \in R$, we define a primed copy r' that refers to the same register in the post-state of a transition, and we define $R' = \{r' \mid r \in R\}$. We use r_1, r_2, r_3 to denote register variables, and x, y, z to denote communication variables. Thus, the free variables in φ are either variables from R describing registers in the pre-state, variables from R' describing registers in the post-state, or a communication variable x. For example, $\mathtt{p} \to \mathtt{q} : x\{even(x) \wedge r_1' = r_1 + 1 \wedge r_2' = x\}$ describes $\mathtt{p}$ sending $\mathtt{q}$ an even number x, incrementing the value of register r_1 by 1, and storing the value of x in register r_2.

4.3 Symbolic Coherence Conditions

We first show that the building blocks of *CC* can be encoded as least and greatest fixpoints over a symbolic protocol's transition relation. Then, we use these building blocks to define Symbolic Coherence Conditions. In the remainder of the section, we fix a symbolic protocol $\mathbb{S} = (S, R, \Delta, s_0, \rho_0, F)$ whose concretization is a GCLTS. Additionally, we define two copies of the symbolic protocol, denoted $\mathbb{S}_1$ and $\mathbb{S}_2$. 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.

The first building block used by all three Coherence Conditions is the definition of simultaneously reachable pairs of protocol states, lifted to the symbolic setting. Given a participant and a pair of control states (s_1, s_2) in $\mathbb{S}$, the predicate $\mathsf{prodreach}_{\mathtt{p}}(s_1, \boldsymbol{r_1}, s_2, \boldsymbol{r_2})$ describes the register assignments corresponding to pairs of concrete states (s_1, ρ_1), (s_2, ρ_2) that are simultaneously reachable in the concretization of $\mathbb{S}$, where $\boldsymbol{r_i}$ are vectors of the registers in R_i ordered according to some fixed total order. We define $\mathsf{prodreach}$ as a least fixpoint as follows.

Definition 12 (Simultaneous reachability in product symbolic protocol)**.** Let $\mathtt{p} \in \mathcal{P}$ be a participant and let $s_1, s_1', s_2, s_2' \in S$. Then,

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

The definition encodes $\mathbb{S}_1$ and $\mathbb{S}_2$ executing the witness u independently. 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 $\mathtt{p}$. 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 $\mathtt{p}$, Send Coherence now checks whether all values x_1 that can be sent to some $\mathtt{q}$ in (s_1, ρ_1) can also be sent from (s_2, ρ_2), modulo following ε transitions to reach the actual state where $\mathtt{p}$ can send to $\mathtt{q}$. We thus need to express ε-reachability. We formalize the dual: the predicate $\mathsf{unreach}^{\varepsilon}_{\mathtt{p},\mathtt{q}}(s_2, \boldsymbol{r_2}, x_1)$ expresses that $\mathtt{p}$ *cannot* reach any state where it may send x_1 to $\mathtt{q}$, by following ε transitions from symbolic state $(s_2, \boldsymbol{r_2})$. This is formulated as a greatest fixpoint as follows:

Definition 13 (ε-unreachability of $\mathtt{p}$ sending x to $\mathtt{q}$)**.** For $\mathtt{p}, \mathtt{q} \in \mathcal{P}$ and $s \in S$, let

$$\begin{aligned}\mathsf{unreach}^{\varepsilon}_{\mathtt{p},\mathtt{q}}(s, \boldsymbol{r}, x) :=_{\nu} \big(&\textstyle\bigwedge_{(s,\, \mathtt{p}\to\mathtt{q}:y\{\varphi\},\, s') \in \Delta} \neg\varphi[x/y]\,\big) \\ &\wedge \big(\textstyle\bigwedge_{\substack{(s,\, \mathtt{r}\to\mathtt{t}:y\{\varphi\},\, s') \in \Delta \\ \mathtt{p}\neq\mathtt{r} \wedge \mathtt{p}\neq\mathtt{t}}} \forall y\, \boldsymbol{r'}.\, \varphi \Rightarrow \mathsf{unreach}^{\varepsilon}_{\mathtt{p},\mathtt{q}}(s', \boldsymbol{r'}, x)\big)\ .\end{aligned}$$

The first conjunct checks that whenever $\mathtt{p}$ reaches a state with an outgoing send transition to $\mathtt{q}$, 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 to define Symbolic Send Coherence.

Definition 14 (Symbolic Send Coherence)**.** A symbolic protocol $\mathbb{S}$ satisfies Symbolic Send Coherence when for each transition $s_1 \xrightarrow{\mathtt{p}\to\mathtt{q}:x_1\{\varphi_1\}} s_1' \in \Delta_1$ and state $s_2 \in S$, the following is valid:

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

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. 9 and 10. Consider the pair of states $(q_1, [c \mapsto 0])$ and $(q_3, [c \mapsto 0])$ which are simultaneously reachable for $\mathtt{r}$ in both protocols. The send transition for $\mathtt{r}$ 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. 9 but not in Fig. 10.

Receive Coherence is conditioned on two simultaneously reachable states $(s_1, \boldsymbol{r_1})$ and $(s_2, \boldsymbol{r_2})$ for a participant $\mathtt{q}$. It checks that if $\mathtt{q}$ can receive x from $\mathtt{p}$ in the first state, $\mathtt{q}$ cannot also receive x as the first message from $\mathtt{p}$ in the second state, in which it can also receive from a different participant $\mathtt{r}$, unless $\mathtt{p}$ sending x causally

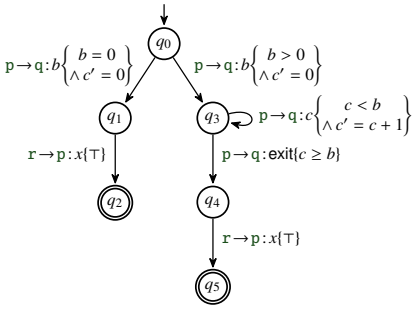

Figure 9: Example where states q_1 and q_3 satisfy Send Coherence for r.

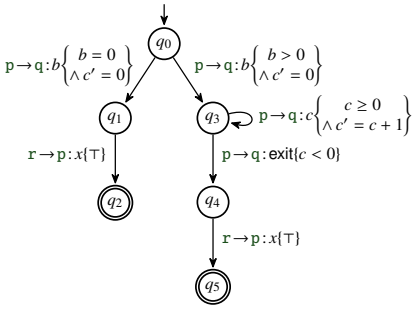

Figure 10: Example where states q_1 and q_3 violate Send Coherence for r.

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 dependencies. We introduce a family of predicates $\mathsf{avail}_{\mathtt{p},\mathtt{q},\mathcal{B}}(x_1, s_2, \boldsymbol{r_2})$ for this purpose. Here, $\mathcal{B}$ tracks the set of participants that are blocked from sending a message because their send action causally depends on q first receiving from r. Note that because $\mathbb{S}$ has finitely many transitions and thus participants, there are finitely many $\mathcal{B}$ sets. The predicates are defined as the least fixpoint of the following mutually recursive definition.

Definition 15 (Symbolic Availability)**.**

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

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). The predicate $\mathsf{avail}_{\mathtt{p},\mathtt{q},\{\mathtt{q}\}}(m, s)$ from Algorithm 1 is defined in terms of the symbolic availability predicate $\mathsf{avail}_{\mathtt{p},\mathtt{q},\{\mathtt{q}\}}(x, s, \boldsymbol{r})$, where $\boldsymbol{r}$ is empty.

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

Definition 16 (Symbolic Receive Coherence)**.** A symbolic protocol $\mathbb{S}$ satisfies Symbolic Receive Coherence when for every pair of transitions $s_1 \xrightarrow{\mathtt{p}\to\mathtt{q}:x_1\{\varphi_1\}} s_1' \in \Delta_1$ and $s_2 \xrightarrow{\mathtt{r}\to\mathtt{q}:x_2\{\varphi_2\}} s_2' \in \Delta_2$ with $\mathtt{p} \neq \mathtt{r}$:

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

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 $\mathtt{p}$.

Definition 17 (Symbolic No Mixed Choice)**.** A symbolic protocol $\mathbb{S}$ satisfies Symbolic No Mixed Choice when for every pair of transitions $s_1 \xrightarrow{\mathtt{p}\to\mathtt{q}:x_1\{\varphi_1\}} s_1' \in \Delta_1$ and $s_2 \xrightarrow{\mathtt{r}\to\mathtt{p}:x_2\{\varphi_2\}} s_2' \in \Delta_2$:

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

The preciseness of our Symbolic Coherence Conditions is stated as follows.

Theorem 18 (Preciseness of Symbolic Coherence Conditions)**.** $\mathbb{S}$ *is implementable if and only if it satisfies Symbolic Send Coherence, Symbolic Receive Coherence, and Symbolic No Mixed Choice.*

4.4 Complexity

Like in §3.5, we use our Symbolic Coherence Conditions to analyze the complexity of implementability for 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 19. *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 [27]. 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 problem of deciding reachability for 1-safe Petri nets [18]. 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. □

5 Extension to Network Architectures

In this section, we motivate how implementability depends on the model of asynchrony, specifically the network architecture of the assumed implementation model. We then introduce network-parametric global protocol semantics, CLTS and the implementability problem, and show how *CC* can be extended to solve the network-parametric setting.

5.1 Motivation

We revisit the non-implementable example from Fig. 1, for which we proposed one method of protocol repair in §3. Observe that, as a specification mechanism, Fig. 1does not a priori commit to a particular network model. While peer-to-peer FIFO communication is ubiquitous in practice, other network models also hold practical and theoretical interest. For example, Erlang and Go implement mailbox communication, and many correctness proofs of classic distributed algorithms such as leader election [21] and clock synchronization [37] rely on bag semantics. We consider the effect of the chosen network architecture in our original analysis. Perhaps surprisingly, replacing the peer-to-peer FIFO network by another asynchronous network architecture does not constitute a viable method of protocol repair. Non-implementability is solely due to the asynchronous nature of communication, specifically, the fact that the two send events $\mathtt{s} \triangleright \mathtt{w}_2!\mathsf{half}$ and $\mathtt{w}_1 \triangleright \mathtt{w}_2!\mathsf{half}$ in the $3c$ run do not causally depend on each other. They can therefore happen concurrently, causing the two messages to arrive at $\mathtt{w}_2$ in any order with arbitrary delays. Thus, the protocol is non-implementable for any asynchronous network architecture.

In general, however, implementability depends on the specific network architecture. To see this, observe that the repair proposed in §3 does not help establish implementability under the mailbox network architecture, in which all messages sent to the same recipient are collected in one FIFO buffer. Consider the execution u_3 under a mailbox network, following branch $3c$.

$$u_3 = \mathtt{s} \triangleright \mathtt{w}_1!\mathsf{full} \cdot \mathtt{w}_1 \triangleleft \mathtt{s}?\mathsf{full} \cdot \mathtt{w}_1 \triangleright \mathtt{w}_2!\mathsf{half} \cdot \mathtt{s} \triangleright \mathtt{w}_2!\mathsf{full}$$

In u_3 the full message sent to $\mathtt{w}_2$ by $\mathtt{s}$ is delayed by network asynchrony, and reordered after the half message sent to $\mathtt{w}_2$ by $\mathtt{w}_1$. Since $\mathtt{w}_2$'s mailbox buffers messages in FIFO order of arrival, this execution forces $\mathtt{w}_2$ to first receive half from $\mathtt{w}_1$ before being able to retrieve the one from $\mathtt{s}$ in the buffer. The issue with mailbox is that in the $3c$ branch, the network may still asynchronously reorder the two messages sent to $\mathtt{w}_2$ by delaying the message from $\mathtt{s}$. Since messages are buffered in FIFO order of arrival, this would force $\mathtt{w}_2$ to first receive the message from $\mathtt{w}_1$ before being able to retrieve the one from $\mathtt{s}$ in the buffer. The resulting execution violates

protocol fidelity. Note that this time, the ensuing violation of implementability has nothing to do with incomplete information by any of the protocol participants about what branch the other participants are following. Instead, it is solely due to the ability of the network architecture to reorder independent events in executions of protocol runs. Moreover, the violation cannot be avoided by changing the candidate local implementation. A possible repair of the protocol for mailbox networks is to introduce a causal dependency between $\mathtt{s} \triangleright \mathtt{w}_2!\mathsf{half}$ and $\mathtt{w}_1 \triangleright \mathtt{w}_2!\mathsf{half}$, e.g., by inserting an additional message exchange $\mathtt{s} \triangleright \mathtt{w}_1!\mathsf{done}$ between the two exchanges, forcing $\mathtt{w}_1$ to wait until $\mathtt{s}$ has sent its message to $\mathtt{w}_2$.

Finally, if we swap the network architecture from mailbox to *mailbag*, where the single FIFO queue per recipient is replaced by an unordered multiset, the protocol becomes implementable again even without the proposed second repair for mailbox.

5.2 Network-Parametric Framework

We generalize our distributed implementation model and global protocol semantics along the axis of network parametricity. First, we formalize the definition of network architectures $\mathbb{A}$.

Definition 20 (Network architecture)**.** A *network architecture* over a set of participants $\mathcal{P}$ and message values $\mathcal{V}$ is a tuple $\mathbb{A} = (\mathsf{Chan}, \mathsf{B}, \mathsf{bf}, \mathsf{ins}, \mathsf{rem}, b_0)$ where Chan is a set of channels, B a set of *channel contents* and $\mathsf{bf} : \mathcal{P} \times \mathcal{P} \to \mathsf{Chan}$ a map that associates each sender and receiver with a channel. Intuitively, $\mathsf{bf}(\mathtt{p}, \mathtt{q})$ denotes the message buffer to which messages sent from $\mathtt{p}$ to $\mathtt{q}$ are deposited. We refer to bf as the *communication topology* of $\mathbb{A}$. The set of *channel states* is $\Xi = \mathsf{Chan} \to \mathsf{B}$. Messages $m \in M$ in channel contents are tagged with their sender and receiver, i.e., $M = \mathcal{P} \times \mathcal{P} \times \mathcal{V}$. Channel contents are equipped with partial insert and remove operations, $\mathsf{ins}, \mathsf{rem} : M \to \mathsf{B} \to \mathsf{B}$, where $\mathsf{ins}(m)(b)$ being undefined indicates that b blocks on inserting m and $\mathsf{rem}(m)(b)$ is only defined when m is available for removal in b. Finally, $b_0 \in \mathsf{B}$ is the empty channel contents.

Example 21. We define eight concrete network architectures that we will revisit later. We consider four communication topologies: n-to-n, in which all senders and receivers share the same channel, one-to-n, in which receivers share the same channel to receive from a single sender, n-to-one, in which senders share the same channel to send to a single receiver, and one-to-one, in which each sender and receiver pair have a unique channel. We consider two message buffer data structures: ordered FIFO queues, and unordered multisets. The aforementioned network architectures often appear under the names of global bus (n-to-n FIFO), message soup (n-to-n multiset), and mailbox (one-to-n FIFO). Message soups are commonly found in leader election protocols such as Paxos and Raft, and one-to-n

FIFO is found in work stealing patterns in parallel programming. Finally, the one-to-one or peer-to-peer FIFO is the standard network architecture for CSMs, and widely assumed in the theory and practice of message-passing concurrency.

The four communication topologies are defined as follows (with our naming conventions given in parenthesis, where "B" refers to the name of one of the buffer types below):

- one-to-one (peer-to-peer B): $\mathsf{Chan} = \mathcal{P} \times \mathcal{P}, \mathsf{bf}(\mathtt{p}, \mathtt{q}) = (\mathtt{p}, \mathtt{q})$,
- one-to-n (senderB): $\mathsf{Chan} = \mathcal{P}, \mathsf{bf}(\mathtt{p}, \mathtt{q}) = \mathtt{p}$,
- n-to-one (mailB): $\mathsf{Chan} = \mathcal{P}, \mathsf{bf}(\mathtt{p}, \mathtt{q}) = \mathtt{q}$,
- n-to-n (monoB): $\mathsf{Chan} = \{0\}, \mathsf{bf}(\mathtt{p}, \mathtt{q}) = 0$,

and the two buffer types are FIFO queues (B=box), $\mathsf{B} = M^*$, and multisets (B=bag), $\mathsf{B} = M \to \mathbb{N}$. In the case of FIFO queues, *insert* corresponds to appending at the end of the queue, *remove* corresponds to removing from the head, and the empty buffer contents b_0 is ε; in the case of multisets, *insert* is multiset addition, *remove* multiset deletion, and $\emptyset$ is the empty buffer.

We note that the four network architectures with homogeneous bag channels are operationally equivalent and collapse to the monobag case: since messages are all labeled with their sender and receiver, one can "on demand" separate the message soup into $\mathcal{P}^2$ multisets, or $\mathcal{P}$ multisets by sender or receiver whenever messages are sent or received, and thus simulate the other network architectures. This leaves us with the four FIFO network architectures, which we refer to as `p2p`, `mb`, `sb`, and `monob` in the rest of the paper, and the collapsed case for bag channels (`bag`).

We generalize CLTS to be parametric in a choice of network architecture $\mathbb{A} = (\mathsf{Chan}, \mathsf{B}, \mathsf{bf}, \mathsf{ins}, \mathsf{rem}, b_0)$. The only change lies in the definition of the CLTS transition relation, in which we replace the hardcoded insert and remove operations for FIFO buffers with *insert* and *remove* defined in terms of ins and rem from $\mathbb{A}$ by lifting ins and rem to channel states $\xi \in \Xi$ of $\mathbb{A}$: we define $\mathit{insert}(\xi, \mathtt{p}, \mathtt{q}, m) = \xi'$ where $\xi'(\mathsf{bf}(\mathtt{p}, \mathtt{q})) = \mathsf{ins}(\mathtt{p}, \mathtt{q}, m)(\xi(\mathsf{bf}(\mathtt{p}, \mathtt{q})))$ and all other channel contents remain the same; we define $\mathit{remove}(\xi, \mathtt{p}, \mathtt{q}, m)$ analogously.

- $(\vec{s}, \xi) \xrightarrow{\mathtt{p} \triangleright \mathtt{q}!m} (\vec{s}', \xi')$ if $(\vec{s}_\mathtt{p}, \mathtt{p} \triangleright \mathtt{q}!m, \vec{s}'_\mathtt{p}) \in \delta_\mathtt{p}$, $\vec{s}_\mathtt{r} = \vec{s}'_\mathtt{r}$ for every participant $\mathtt{r} \neq \mathtt{p}$, $\xi' = \mathit{insert}(\xi, \mathtt{p}, \mathtt{q}, m)$.
- $(\vec{s}, \xi) \xrightarrow{\mathtt{q} \triangleleft \mathtt{p}?m} (\vec{s}', \xi')$ if $(\vec{s}_\mathtt{q}, \mathtt{q} \triangleleft \mathtt{p}?m, \vec{s}'_\mathtt{q}) \in \delta_\mathtt{q}$, $\vec{s}_\mathtt{r} = \vec{s}'_\mathtt{r}$ for every participant $\mathtt{r} \neq \mathtt{q}$, $\xi' = \mathit{remove}(\xi, \mathtt{p}, \mathtt{q}, m)$.

For example, in the CLTS with $T_\mathtt{s}, T_{\mathtt{w}_1}, T_{\mathtt{w}_2}$ defined by Figs. 2 to 4 in §1, and $\mathbb{A} = \mathtt{mb}$, the configuration reached on u_3 is a deadlock, whereas in the same CLTS but with $\mathbb{A} = \mathtt{p2p}$ or $\mathbb{A} = \mathtt{bag}$, the configuration reached on u_3 is not a deadlock.

Having defined $\mathbb{A}$ formally, the definition of channel compliance from §2 can now be instantiated for different network architectures beyond peer-to-peer FIFO. For example, u_1, u_2 from §3and u_3 above are $\mathbb{A}$-channel-compliant for all $\mathbb{A} = \mathtt{p2p}, \mathtt{mb}, \mathtt{sb}, \mathtt{monob}, \mathtt{bag}$. The prefix $\mathtt{s} \triangleright \mathtt{w}_1!\mathsf{full} \cdot \mathtt{w}_1 \triangleleft \mathtt{s}?\mathsf{full}$ of u_3 is channel-matched, $\mathtt{w}_1 \triangleleft \mathtt{s}?\mathsf{full} \cdot \mathtt{s} \triangleright \mathtt{w}_1!\mathsf{full}$ is not $\mathbb{A}$-channel-compliant for any choice of $\mathbb{A}$, and $u_3 \cdot \mathtt{w}_2 \triangleleft \mathtt{s}?\mathsf{full}$ is $\mathtt{p2p}$-channel-compliant but not $\mathtt{mb}$- or $\mathtt{monob}$-channel-compliant.

Our global protocol semantics is thus already network-parametric, as $\mathbb{A}$ is included as a parameter to the definition provided in §2. Defining the network-parametric implementability problem is likewise straightforward:

Definition 22 (Network-Parametric Protocol Implementability)**.** A protocol $\mathcal{S}$ is *implementable* under network architecture $\mathbb{A}$ if there exists a CLTS $\mathcal{T}_{\mathbb{A}} = \{T_{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}}$ such that the following two properties hold: (i) *protocol fidelity*: $\mathcal{L}(\{T_{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}}) = C^{\sim}_{\mathbb{A}}(\mathcal{S})$, and (ii) *deadlock freedom*: $\{T_{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}}$ is deadlock-free. We say that $\{T_{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}}$ implements $\mathcal{S}$ under $\mathbb{A}$.

5.3 Network-Parametric Coherence Conditions

Next, we present the generalization of *CC* to the network-parametric setting. We focus on the delta between *CC* for $\mathtt{p2p}$ and for the network-parametric setting, which consists of a modular modification of Receive Coherence, and the addition of a brand new condition, Prefix Extensibility. Send Coherence and No Mixed Choice remain sound and complete in their original form across all network architectures.

Generalized Receive Coherence. The key observation is that irrespective of network architecture, it is sufficient and necessary to guarantee that no two distinct receptions are simultaneously enabled for any participant from any configuration. Furthermore, the notion of a message m being receivable by $\mathtt{q}$ from $\mathtt{p}$ after executing a word w can be captured in a network-parametric way, as the channel compliance of $w \cdot \mathtt{q} \triangleleft \mathtt{p}?m$.

Definition 23 (Generalized Receive Coherence [47])**.** A protocol $\mathcal{S}$ satisfies *Generalized Receive Coherence* (GRC) when for every two simultaneously reachable states for $\mathtt{q}$, s_1 and s_2, with transitions $s_1 \xrightarrow{\mathtt{p}\to\mathtt{q}:m_1} s_1'$ and $s_2 \xrightarrow{\mathtt{r}\to\mathtt{q}:m_2} s_2'$, if $\mathtt{r} \neq \mathtt{p}$ or $m_2 \neq m_1$, then for any $w \in \mathrm{pref}(\mathcal{L}(\mathcal{S}_{s_2}))$ such that $w{\Downarrow_{\Sigma_{\mathtt{q}}}} = \varepsilon$ and $\mathtt{r} \triangleright \mathtt{q}!m_2 \leq w{\Downarrow_{\Sigma_{\mathtt{r}}}}$, the extension $w \cdot \mathtt{q} \triangleleft \mathtt{p}?m_1$ is not channel-compliant.

Generalized Receive Coherence differs from RC for $\mathtt{p2p}$ in two ways: w is a protocol prefix from s_2 rather than s_2', and the condition requires an explicit send from $\mathtt{r}$ to appear in w. For $\mathtt{p2p}$ channel compliance, the two formulations are equivalent because per-sender channels are independent; retaining the more general form allows handling $\mathtt{mb}$ and $\mathtt{monob}$ without ruling them out a priori.

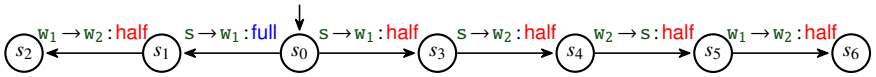

Figure 11: Subprotocol of task scheduler from Fig. 1, containing parts of branch 1*a* (left) and branch 3*c* (right).

Prefix Extensibility. Our new condition, called Prefix Extensibility, captures a source of non-implementability that never arises in `p2p` networks but arises in other architectures. To illustrate this new source of non-implementability, consider the simple straight-line global specification consisting only of the synchronous word $\mathtt{p}_1 \to \mathtt{q} : m \cdot \mathtt{p}_2 \to \mathtt{q} : m$. Despite its simplicity, there does not exist a deadlock-free `mb` (or `monob`) CLTS that implements this specification. Any candidate CLTS must exhibit the following deadlocking trace: $\mathtt{p}_2 \triangleright \mathtt{q}!m \cdot \mathtt{p}_1 \triangleright \mathtt{q}!m$. Because the network can reorder $\mathtt{p}_2$'s send event before $\mathtt{p}_1$'s send event, yet the local actions of $\mathtt{q}$ tell it to receive in the opposite order, the only way for the CLTS to not deadlock is for $T_{\mathtt{q}}$ to admit the local trace $\mathtt{q} \triangleleft \mathtt{p}_2?m \cdot \mathtt{q} \triangleleft \mathtt{p}_1?m$.

To rule out such cases of non-implementability, Prefix Extensibility requires that protocol prefixes are closed under per-participant equality.

Definition 24 (Prefix Extensibility [47])**.** A protocol $\mathcal{S}$ satisfies *Prefix Extensibility* (PE) under $\mathbb{A}$ when for every $\rho \in \mathrm{pref}(\mathcal{L}(\mathcal{S}))$ and every $\mathbb{A}$-channel-compliant $w \in \Sigma^*_{async}$ that agrees with ρ per participant, there exists $u \in \Sigma^*_{async}$ such that wu is channel-compliant and $wu \equiv_{\mathcal{P}} \mathtt{split}(\rho)$.

Prefix Extensibility holds trivially for *any* protocol under `p2p`, `sb`, and `bag`: for these architectures, any channel-compliant word agreeing with a synchronous trace can be extended to complete that trace while preserving channel compliance [47]. For `mb` and `monob`, Prefix Extensibility must be checked on a given protocol's runs.

In Fig. 11, depicting a subprotocol of the task scheduling protocol from Fig. 1, the protocol run corresponding to the bottom branch violates Prefix Extensibility.

Our network-parametric characterization of implementability is thus defined as the conjunction of four conditions.

Definition 25 (Generalized Coherence Conditions)**.** A protocol $\mathcal{S}$ satisfies Generalized Coherence Conditions under network architecture $\mathbb{A}$ when it satisfies Send Coherence, Generalized Receive Coherence, No Mixed Choice, and Prefix Extensibility.

5.4 Channel Compliance Axioms

In this section, we provide sufficient conditions under which our Generalized Coherence Conditions are precise for a given network architecture. These conditions

describe network architectures, which we define semantically as channel compliance definitions, and thus we refer to the conditions as *channel compliance axioms*. A network architecture $\mathbb{A} \in \mathfrak{A}$ satisfies our axiomatic network model if $\mathbb{A}$ satisfies **F1** through **F6**:

Definition 26 (Channel compliance axioms)**.** Let $w \in \Sigma^*_{async}$ be channel-compliant.

- **F1** For all $x \in \Sigma_!$, wx is channel-compliant.
- **F2** For all $\mathtt{p} \neq \mathtt{q} \in \mathcal{P}$ and $m \in \mathcal{V}$, $|w{\Downarrow_{\mathtt{p}\triangleright\mathtt{q}!m}}| \geq |w{\Downarrow_{\mathtt{q}\triangleleft\mathtt{p}?m}}|$.
- **F3** For all $\rho \in \Gamma^*_{sync}$, $\mathtt{split}(\rho)$ is channel-compliant.
- **F4** For all $\rho \in \Gamma^*_{sync}$, if $w \equiv_{\mathcal{P}} \mathtt{split}(\rho)$, then w is channel-matched.
- **F5** For all $x \in \Sigma_!$, $y \in \Sigma_?$, if wy is channel-compliant then wxy is channel-compliant.
- **F6** Let $\alpha, \beta \in \Gamma^*_{sync}$, $\mathtt{p} \neq \mathtt{q} \in \mathcal{P}$ and $m \in \mathcal{V}$ such that for all $\mathtt{p} \in \mathcal{P}$, $w{\Downarrow_{\Sigma_\mathtt{p}}} \leq \mathtt{split}(\alpha\beta){\Downarrow_{\Sigma_\mathtt{p}}}$, and $w{\Downarrow_{\Sigma_\mathtt{q}}} = \mathtt{split}(\alpha){\Downarrow_{\Sigma_\mathtt{q}}}$, and $w \cdot \mathtt{q}\triangleleft\mathtt{p}?m$ is channel-compliant. Then, there exists w' such that w' is compliant with β, $w'{\Downarrow_{\Sigma_\mathtt{q}}} = \varepsilon$ and $w' \cdot \mathtt{q} \triangleleft \mathtt{p}?m$ is channel-compliant.

FIFO queues and multisets both satisfy all eight axioms. Therefore, all five considered architectures are in $\mathfrak{A}$. Heterogeneous architectures mixing FIFO and bag channels also satisfy the axioms. However, priority queues, channels with duplication, and bounded channels are ruled out.

The following theorem states that our Generalized Coherence Conditions are sound and complete with respect to implementability, assuming a network architecture that satisfies our channel compliance axioms. The proof of Theorem 27 is fully mechanized in Rocq, and builds on the mechanization of Theorem 5.

Theorem 27 (Preciseness of Generalized Coherence Conditions)**.** *Let $\mathbb{A}$ be a network architecture satisfying* **F1** *through* **F6** *and let $\mathcal{S}$ be a global protocol. Then, $\mathcal{S}$ is implementable under $\mathbb{A}$ if and only if it satisfies Generalized Coherence Conditions under $\mathbb{A}$.*

5.5 Derived Results

The matching complexity bounds, synthesis result, and generalization to infinite-state protocols via μCLP encodings all transfer to the network-parametric setting. Here, we briefly discuss selected takeaways.

First, the network-parametric generalization reinforces the separation between synthesis and implementability: once implementability is established, the same

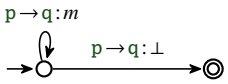

Figure 12: A protocol $\mathcal{S}_a$ that is not bag-implementable but p2p-implementable.

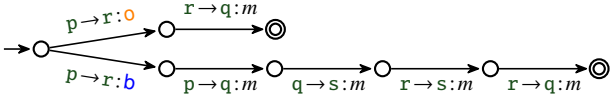

Figure 13: A protocol $\mathcal{S}_b$ that is not p2p-implementable but sb-implementable.

canonical implementation suffices to implement it. Second, the μCLP encoding of Generalized Receive Coherence yields the following relationships between the classes of implementable global protocols under each network architecture. The strictness is witnessed by the protocols $\mathcal{S}_a$ and $\mathcal{S}_b$ depicted in Figs. 12 to 13.

Lemma 28 (Implementability relationships)**.** *Any* bag*-implementable protocol is* p2p*-implementable, and any* p2p*-implementable protocol is* sb*-implementable. Both inclusions are strict.*

Third, on complexity, the hardness reduction relies on program order and not network reorderings for Receive Coherence, and thus the same lower bound construction with a small modification works for the remaining network architectures. In order to show that the construction carries over to bag and senderbox networks, one is only required to analyze the unique message receptions that appear in each gadget. Similarly, the directed choice case falls out for free.

Finally, we conclude this section by mentioning that [47] takes network parametricity a step further by reducing the channel compliance axioms to *buffer axioms*, which allows one to obtain heterogeneous channel architectures for free, as well as not having to reprove the channel compliance axioms from e.g. senderbox to mailbox. As this result is not included in the author's thesis, it is omitted here.

6 Tool Implementation and Mechanization

6.1 The Sprout Tool

All decision procedures for implementability discussed in this paper are implemented in an open-source tool called Sprout, which is hosted at `https://github.com/nyu-acsys/sprout`. We refer the reader to [45] for details about Sprout's implementation, including optimizations to the μCLP encodings presented in §4 that make validity checking tractable for Sprout's backend solver MuVal. [45] also describes μCLP encodings for checking whether a symbolic protocol falls into the GCLTS fragment. We additionally refer the reader to Sprout's source code for a comprehensive benchmark suite of communication protocols drawn from the MST literature.

6.2 Rocq Mechanization

We conclude by discussing a subtle bug in the infinite-word semantics of global protocols uncovered during the mechanization of Theorem 5, which originated from [49] and is present in several subsequent works [41, 42] and preprints [44]. The bug resulted in a definition of global protocol semantics that is spuriously non-implementable due to selectively excluding infinite traces that are indistinguishable to a CLTS. We say that two words $w, w' \in \Sigma^{\omega}_{async}$ are *indistinguishable* to a CLTS when w is a CLTS trace iff w' is a CLTS trace. Note that indistinguishability is network-specific: two words that are `p2p`-indistinguishable may not be `mb`-indistinguishable.

Indistinguishability of pairs of finite words is straightforward to define, given that w and w' must be permutations of each other, and moreover, the total order of events for each participant must be identical. Thus, to show that *any* CLTS executes w', we only require that each participant's local implementation accepts $w{\Downarrow_{\Sigma_{\mathtt{p}}}}$, which is given from the fact that the CLTS recognizes w. For infinite words, however, indistinguishability can no longer be defined purely alphabetically, due to the possibility for a CLTS to infinitely reorder independent events, achieving the effect of indefinite delay. For example, the pair of infinite words $v_1 = \mathtt{p} \triangleright \mathtt{q}!m^{\omega}$ and $v_2 = \mathtt{r} \triangleright \mathtt{s}!m \cdot \mathtt{p} \triangleright \mathtt{q}!m^{\omega}$ are not indistinguishable in general because v_1 provides no information about the local implementation of participant `r`, yet to know that v_2 is a trace of a given CLTS requires us to know that participant `r`'s local implementation recognizes the trace $\mathtt{r} \triangleright \mathtt{s}!m$.

We show by counterexample that the flawed definition of infinite protocol semantics, given below, is not closed under CLTS indistinguishability of infinite words.

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

Consider the simple protocol depicted in Fig. 14, involving four participants `p`, `q`, `r`, `s` and two message values m, m'. Fig. 15 depicts a protocol $\mathcal{S}_d$ obtained by a simple state renaming of $\mathcal{S}_c$ that is semantically identical to $\mathcal{S}_c$. Clearly, both protocols are implementable, and moreover by the same canonical CLTS, which admits the infinite trace $\overline{v} = \mathtt{r} \triangleright \mathtt{s}!m \cdot \mathtt{p} \triangleright \mathtt{q}!m^{\omega}$. We should thus expect that $\overline{v}$ is included in the infinite semantics of both $\mathcal{S}_c$ and $\mathcal{S}_d$. Unfortunately, per the above definition, $\overline{v}$ is included in $\overline{C}^{\sim}_{inf}(\mathcal{S}_d)$ but excluded by $\overline{C}^{\sim}_{inf}(\mathcal{S}_c)$: in $\mathcal{S}_c$, there exists no infinite run that includes any events by `r` or `s`, and thus no existential witness for w that satisfies the subsequent conjuncts.

Fortunately, the correction amounts to simply swapping the first two quantifiers in $\overline{C}^{\sim}_{inf}(\mathcal{S})$, see the definition in §2. The correct definition weakens the requirement that all finite prefixes of a infinite CLTS trace must be consistent with a *single*

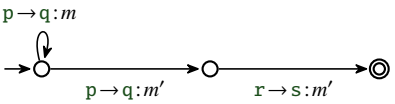

Figure 14: Example protocol with infinite words $\mathcal{S}_c$.

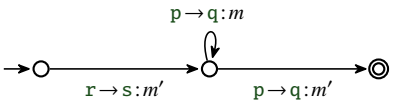

Figure 15: Example protocol with infinite words $\mathcal{S}_d$.

infinite global protocol run. Instead, each finite prefix may be consistent with a different finite run *prefix*, which may in turn be part of a finite or infinite maximal run in $\mathcal{S}$. The revised definition more closely matches simulation-based notions of trace equivalence, for example in [75], and incurred minimal requisite changes to proofs. We refer the reader to [46] for more details.

Funding Acknowledgement. This work was supported by the National Science Foundation under grant agreement 2304758.

References

[1] Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013. pp. 1–8. IEEE (2013), https://ieeexplore.ieee.org/document/6679385/

[2] Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. IEEE Trans. Software Eng. **29**(7), 623–633 (2003), https://doi.org/10.1109/TSE.2003.1214326

[3] Bertrand, N., Bouyer, P., Brihaye, T., Carlier, P.: When are stochastic transition systems tameable? J. Log. Algebraic Methods Program. **99**, 41–96 (2018), https://doi.org/10.1016/j.jlamp.2018.03.004

[4] Bertrand, N., Stainer, A., Jéron, T., Krichen, M.: A game approach to determinize timed automata. Formal Methods Syst. Des. **46**(1), 42–80 (2015), https://doi.org/10.1007/s10703-014-0220-1

[5] Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (1983), https://doi.org/10.1145/322374.322380

[6] Castro-Perez, D., Hu, R., Jongmans, S., Ng, N., Yoshida, N.: Distributed programming using role-parametric session types in go: statically-typed endpoint apis for dynamically-instantiated communication structures. Proc. ACM Program. Lang. **3**(POPL), 29:1–29:30 (2019), https://doi.org/10.1145/3290342

[7] Castro-Perez, D., Yoshida, N.: Dynamically updatable multiparty session protocols: Generating concurrent go code from unbounded protocols. In: Ali, K., Salvaneschi, G. (eds.) 37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States. LIPIcs, vol. 263, pp. 6:1–6:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023), https://doi.org/10.4230/LIPIcs.ECOOP.2023.6

[8] Chaudhuri, S., Ellis, K., Polozov, O., Singh, R., Solar-Lezama, A., Yue, Y.: Neurosymbolic programming. Found. Trends Program. Lang. **7**(3), 158–243 (2021). https://doi.org/10.1561/2500000049

[9] Church, A.: Applications of recursive arithmetic to the problem of circuit synthesis. In: Summaries of Talks Presented at the Summer Institute for Symbolic Logic. pp. 3–50. Cornell University (1957)

[10] Clemente, L., Lasota, S., Piórkowski, R.: Determinisability of register and timed automata. Log. Methods Comput. Sci. **18**(2) (2022), https://doi.org/10.46298/lmcs-18(2:9)2022

[11] Coppo, M., Dezani-Ciancaglini, M., Padovani, L., Yoshida, N.: A gentle introduction to multiparty asynchronous session types. In: Bernardo, M., Johnsen, E.B. (eds.) 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, pp. 146–178. Springer (2015), https://doi.org/10.1007/978-3-319-18941-3_4

[12] Cruz-Filipe, L., Montesi, F.: Choreographies in practice. In: Albert, E., Lanese, I. (eds.) 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. Lecture Notes in Computer Science, vol. 9688, pp. 114–123. Springer (2016), https://doi.org/10.1007/978-3-319-39570-8_8

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

[14] Cutner, Z., Yoshida, N., Vassor, M.: Deadlock-free asynchronous message reordering in rust with multiparty session types. In: Lee, J., Agrawal, K., Spear, M.F. (eds.) PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022. pp. 246–261. ACM (2022), https://doi.org/10.1145/3503221.3508404

[15] D'Antoni, L., Veanes, M.: The power of symbolic automata and transducers. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part

I. Lecture Notes in Computer Science, vol. 10426, pp. 47–67. Springer (2017), https://doi.org/10.1007/978-3-319-63387-9_3

[16] Demangeon, R., Honda, K., Hu, R., Neykova, R., Yoshida, N.: Practical interruptible conversations: distributed dynamic verification with multiparty session types and python. Formal Methods Syst. Des. **46**(3), 197–225 (2015), https://doi.org/10.1007/s10703-014-0218-8

[17] Ellul, K., Krawetz, B., Shallit, J.O., Wang, M.: Regular expressions: New results and open problems. J. Autom. Lang. Comb. **10**(4), 407–437 (2005), https://doi.org/10.25596/jalc-2005-407

[18] Esparza, J., Nielsen, M.: Decidability issues for petri nets - a survey. J. Inf. Process. Cybern. **30**(3), 143–160 (1994)

[19] Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G.C., Larus, J.R., Levi, S.: Language support for fast and reliable message-based communication in singularity OS. In: Berbers, Y., Zwaenepoel, W. (eds.) Proceedings of the 2006 EuroSys Conference, Leuven, Belgium, April 18-21, 2006. pp. 177–190. ACM (2006), https://doi.org/10.1145/1217935.1217953

[20] Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: 20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings. pp. 321–330. IEEE Computer Society (2005), https://doi.org/10.1109/LICS.2005.53

[21] Gallager, R.G., Humblet, P.A., Spira, P.M.: A distributed algorithm for minimum-weight spanning trees. ACM Trans. Program. Lang. Syst. **5**(1), 66–77 (1983), https://doi.org/10.1145/357195.357200

[22] Gancher, J., Gibson, S., Singh, P., Dharanikota, S., Parno, B.: Owl: Compositional verification of security protocols via an information-flow type system. In: 44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023. pp. 1130–1147. IEEE (2023), https://doi.org/10.1109/SP46215.2023.10179477

[23] Gazagnaire, T., Genest, B., Hélouët, L., Thiagarajan, P.S., Yang, S.: Causal message sequence charts. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4703, pp. 166–180. Springer (2007), https://doi.org/10.1007/978-3-540-74407-8_12

[24] Genest, B., Muscholl, A.: 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. pp. 2–4. IEEE Computer Society (2005), https://doi.org/10.1109/ACSD.2005.25

[25] Genest, B., Muscholl, A., Peled, D.A.: Message sequence charts. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) 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, pp. 537–558. Springer (2003), https://doi.org/10.1007/978-3-540-27755-2_15

[26] Giallorenzo, S., Montesi, F., Peressotti, M.: Choral: Object-oriented choreographic programming. ACM Trans. Program. Lang. Syst. **46**(1), 1:1–1:59 (2024), https://doi.org/10.1145/3632398

[27] Godefroid, P., Yannakakis, M.: Analysis of boolean programs. In: Piterman, N., Smolka, S.A. (eds.) 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, pp. 214–229. Springer (2013), https://doi.org/10.1007/978-3-642-36742-7_16

[28] Honda, K., Marques, E.R.B., Martins, F., Ng, N., Vasconcelos, V.T., Yoshida, N.: Verification of MPI programs using session types. In: Träff, J.L., Benkner, S., Dongarra, J.J. (eds.) 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, pp. 291–293. Springer (2012), https://doi.org/10.1007/978-3-642-33518-1_37

[29] Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 273–284. ACM (2008), https://doi.org/10.1145/1328438.1328472

[30] Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: Stevens, P., Wasowski, A. (eds.) 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, pp. 401–418. Springer (2016), https://doi.org/10.1007/978-3-662-49665-7_24

[31] Hu, R., Yoshida, N.: Explicit connection actions in multiparty session types. In: Huisman, M., Rubin, J. (eds.) 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, pp. 116–133. Springer (2017), https://doi.org/10.1007/978-3-662-54494-5_7

[32] Imai, K., Neykova, R., Yoshida, N., Yuen, S.: Multiparty session programming with global protocol combinators. In: Hirschfeld, R., Pape, T. (eds.) 34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17,

2020, Berlin, Germany (Virtual Conference). LIPIcs, vol. 166, pp. 9:1–9:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020), https://doi.org/10.4230/LIPIcs.ECOOP.2020.9

[33] International Telecommunication Union: ITU-T Recommendation Z.120: Message Sequence Chart (MSC). ITU-T Recommendation Z.120, International Telecommunication Union, Geneva (February 2011), https://www.itu.int/rec/T-REC-Z.120-201102-I/en

[34] Kashiwa, S., Shen, G., Zare, S., Kuper, L.: Portable, efficient, and practical library-level choreographic programming (2023), https://arxiv.org/abs/2311.11472

[35] Kupferman, O., Vardi, M.Y.: Synthesizing distributed systems. In: 16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings. pp. 389–398. IEEE Computer Society (2001), https://doi.org/10.1109/LICS.2001.932514

[36] Lagaillardie, N., Neykova, R., Yoshida, N.: Stay safe under panic: Affine rust programming with multiparty session types (artifact). Dagstuhl Artifacts Ser. **8**(2), 09:1–09:16 (2022), https://doi.org/10.4230/DARTS.8.2.9

[37] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. In: Concurrency: the Works of Leslie Lamport, pp. 179–196 (2019), https://doi.org/10.1145/3335772.3335934

[38] Lange, J., Ng, N., Toninho, B., Yoshida, N.: A static verification framework for message passing in go using behavioural types. In: Chaudron, M., Crnkovic, I., Chechik, M., Harman, M. (eds.) Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. pp. 1137–1148. ACM (2018), https://doi.org/10.1145/3180155.3180157

[39] Languages, Systems, and Data Lab, UC Santa Cruz: Chorus: Choreographic programming in rust. https://lsd-ucsc.github.io/ChoRus/introduction.html, accessed: 2025-07-10

[40] Li, E.: Decision Problems for Global Protocol Specifications. Ph.D. thesis (2025), https://www.proquest.com/dissertations-theses/decision-problems-global-protocol-specifications/docview/3255120134/se-2, copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2025-12-06

[41] Li, E., Stutz, F., Wies, T.: Deciding subtyping for asynchronous multiparty sessions. In: Weirich, S. (ed.) 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, pp. 176–205. Springer (2024), https://doi.org/10.1007/978-3-031-57262-3_8

[42] Li, E., Stutz, F., Wies, T., Zufferey, D.: Complete multiparty session type projection with automata. In: Enea, C., Lal, A. (eds.) Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III. Lecture Notes in Computer Science, vol. 13966, pp. 350–373. Springer (2023), https://doi.org/10.1007/978-3-031-37709-9_17

[43] Li, E., Stutz, F., Wies, T., Zufferey, D.: Characterizing implementability of global protocols with infinite states and data. Proc. ACM Program. Lang. **9**(OOPSLA1), 1434–1463 (2025), https://doi.org/10.1145/3720493

[44] Li, E., Stutz, F., Wies, T., Zufferey, D.: Characterizing implementability of global protocols with infinite states and data (2025), https://arxiv.org/abs/2411.05722

[45] Li, E., Stutz, F., Wies, T., Zufferey, D.: Sprout: A verifier for symbolic multiparty protocols (2025), https://doi.org/10.1007/978-3-031-98682-6, 1st edition, ISBN for softcover and eBook

[46] Li, E., Wies, T.: Certified implementability of global multiparty protocols. In: Forster, Y., Keller, C. (eds.) 16th International Conference on Interactive Theorem Proving, ITP 2025, Reykjavik, Iceland, September 28 - October 1, 2025. pp. 15:1–15:20. LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2025), https://doi.org/10.4230/LIPIcs.ITP.2025.15

[47] Li, E., Wies, T.: Implementability of global distributed protocols modulo network architectures. CoRR **abs/2602.10320** (2026), https://doi.org/10.48550/arXiv.2602.10320

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

[49] Majumdar, R., Mukund, M., Stutz, F., Zufferey, D.: Generalising projection in asynchronous multiparty session types. In: Haddad, S., Varacca, D. (eds.) 32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference. LIPIcs, vol. 203, pp. 35:1–35:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021), https://doi.org/10.4230/LIPIcs.CONCUR.2021.35

[50] Majumdar, R., Pirron, M., Yoshida, N., Zufferey, D.: Motion session types for robotic interactions (brave new idea paper). In: Donaldson, A.F. (ed.) 33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom. LIPIcs, vol. 134, pp. 28:1–28:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019), https://doi.org/10.4230/LIPIcs.ECOOP.2019.28

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

[52] Mauw, S., Reniers, M.A.: High-level message sequence charts. In: Cavalli, A.R., Sarma, A. (eds.) SDL '97 Time for Testing, SDL, MSC and Trends - 8th International SDL Forum, Evry, France, 23-29 September 1997, Proceedings. pp. 291–306. Elsevier (1997)

[53] Montesi, F.: Introduction to Choreographies. Cambridge University Press (2023). https://doi.org/10.1017/9781108981491

[54] de Muijnck-Hughes, J., Vanderbauwhede, W.: A Typing Discipline for Hardware Interfaces. In: Donaldson, A.F. (ed.) 33rd European Conference on Object-Oriented Programming (ECOOP 2019). Leibniz International Proceedings in Informatics (LIPIcs), vol. 134, pp. 6:1–6:27. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2019), https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2019.6

[55] Neykova, R., Bocchi, L., Yoshida, N.: Timed runtime monitoring for multiparty conversations. Formal Aspects Comput. **29**(5), 877–910 (2017), https://doi.org/10.1007/s00165-017-0420-8

[56] Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A session type provider: compile-time API generation of distributed protocols with refinements in f#. In: Dubach, C., Xue, J. (eds.) Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria. pp. 128–138. ACM (2018), https://doi.org/10.1145/3178372.3179495

[57] Neykova, R., Yoshida, N.: Multiparty session actors. Log. Methods Comput. Sci. **13**(1) (2017), https://doi.org/10.23638/LMCS-13(1:17)2017

[58] Ng, N., Yoshida, N., Honda, K.: Multiparty session C: safe parallel programming with message optimisation. In: Furia, C.A., Nanz, S. (eds.) Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7304, pp. 202–218. Springer (2012), https://doi.org/10.1007/978-3-642-30561-0_15

[59] Niu, X., Ng, N., Yuki, T., Wang, S., Yoshida, N., Luk, W.: EURECA compilation: Automatic optimisation of cycle-reconfigurable circuits. In: Ienne, P., Najjar, W.A., Anderson, J.H., Brisk, P., Stechele, W. (eds.) 26th International Conference on Field Programmable Logic and Applications, FPL 2016, Lausanne, Switzerland, August 29 - September 2, 2016. pp. 1–4. IEEE (2016), https://doi.org/10.1109/FPL.2016.7577359

[60] Object Management Group: Unified Modeling Language (UML) Website. https://www.uml.org/, accessed: 2025-07-10

[61] Palma, G.D., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: Towards a function-as-a-service choreographic programming language: Examples and applications (2024), https://arxiv.org/abs/2406.09099

[62] Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming

Languages, Austin, Texas, USA, January 11-13, 1989. pp. 179–190. ACM Press (1989), https://doi.org/10.1145/75277.75293

[63] Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: Ausiello, G., Dezani-Ciancaglini, M., Rocca, S.R.D. (eds.) Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings. pp. 652–671. Lecture Notes in Computer Science, Springer (1989), https://doi.org/10.1007/BFb0035790

[64] Roychoudhury, A., Goel, A., Sengupta, B.: Symbolic message sequence charts. ACM Trans. Softw. Eng. Methodol. **21**(2), 12:1–12:44 (2012), https://doi.org/10.1145/2089116.2089122

[65] Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: Müller, P. (ed.) 31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain. LIPIcs, vol. 74, pp. 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017), https://doi.org/10.4230/LIPIcs.ECOOP.2017.24

[66] Shen, G., Kashiwa, S., Kuper, L.: Haschor: Functional choreographic programming for all (functional pearl). CoRR **abs/2303.00924** (2023), https://doi.org/10.48550/arXiv.2303.00924

[67] Solar-Lezama, A.: Program Synthesis by Sketching. Ph.D. thesis, University of California, Berkeley (2008), http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html

[68] Toninho, B., Yoshida, N.: Certifying data in multiparty session types. J. Log. Algebraic Methods Program. **90**, 61–83 (2017), https://doi.org/10.1016/j.jlamp.2016.11.005

[69] Unno, H., Terauchi, T., Gu, Y., Koskinen, E.: Modular primal-dual fixpoint logic solving for temporal verification. Proc. ACM Program. Lang. **7**(POPL), 2111–2140 (2023), https://doi.org/10.1145/3571265

[70] Veanes, M., Bjørner, N.S.: Symbolic automata: The toolkit. In: Flanagan, C., König, B. (eds.) 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, pp. 472–477. Springer (2012), https://doi.org/10.1007/978-3-642-28756-5_33

[71] Veanes, M., de Halleux, P., Tillmann, N.: Rex: Symbolic regular expression explorer. In: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010. pp. 498–507. IEEE Computer Society (2010), https://doi.org/10.1109/ICST.2010.15

[72] Web Services Choreography Working Group: Web services choreography description language version 1.0. W3c candidate recommendation, World Wide Web Consortium (W3C) (November 2005), available at http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/

[73] Yoshida, N.: Programming language implementations with multiparty session types. In: de Boer, F.S., Damiani, F., Hähnle, R., Johnsen, E.B., Kamburjan, E. (eds.) Active Object Languages: Current Research Trends, Lecture Notes in Computer Science, vol. 14360, pp. 147–165. Springer (2024), https://doi.org/10.1007/978-3-031-51060-1_6

[74] Yoshida, N., Hu, R., Neykova, R., Ng, N.: The scribble protocol language. In: Abadi, M., Lluch-Lafuente, A. (eds.) 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, pp. 22–41. Springer (2013), https://doi.org/10.1007/978-3-319-05119-2_3

[75] Zhou, F., Ferreira, F., Hu, R., Neykova, R., Yoshida, N.: Statically verified refinements for multiparty protocols. Proc. ACM Program. Lang. **4**(OOPSLA), 148:1–148:30 (2020), https://doi.org/10.1145/3428216