Forwarders as Process Compatibility, Logically

Marco Carbone¹, Sonia Marin², and Carsten Schürmann¹

¹ Computer Science Department, IT University of Copenhagen, Denmark
carbonem | carsten@itu.dk
² School of Computer Science, University of Birmingham, United Kingdom
s.marin@bham.ac.uk

Session types, originally proposed by Honda et al. [8], are type annotations that ascribe protocols to processes in a concurrent system and determine how they behave when communicating together. Binary session types found a logical justification in linear logic, identified by Caires and Pfenning [2] and later by Wadler [13, 14], which establishes the following correspondences: linear logic propositions as session types, proofs as processes, cut reductions in linear logic as reductions in the operational semantics, and duality as a notion of compatibility ensuring that two processes communications match.

The situation is not as direct for multiparty session types [9, 10], which generalize binary session types to protocols with more than two participants. One of the central observations is that compatibility requires a stronger property than duality, still ensuring that all messages sent by any participating party will eventually be collected by another. In [6], Denielou and Yoshida first proposed the notion of multiparty compatibility as a “new extension to multiparty interactions of the duality condition for binary session type”.

In this work, we carve out a fragment of classical linear logic (CLL) that adequately captures the notion of multiparty compatibility. We observe moreover that the processes corresponding to proofs in this fragment act as forwarders of messages between participants, generalizing existing proposals, such coherence [5], arbiters [3], and medium processes [1]. Our main result is that the multiparty compatibility of processes can be formalized by the means of a forwarder, and conversely, that any set of processes modelled by a forwarder is compatible.

Processes and types. The language of processes we use to represent communicating entities is a standard variant of the π-calculus [12] used in the context of session types: \( P, Q ::= x \leftrightarrow y \)(link) | \( x().P \)(wait) | \( x[] \)(close) | \( x(y).P \)(input) | \( x[y \triangleright P], Q \)(output).

Types, taken to be propositions of CLL, denote the way an endpoint \( x \)(a channel’s end) must be used at runtime. In this work, we must annotate each operator with another endpoint \( u \)(or a list of endpoints \( ˜u \)) in the case of gathering of communications) to make explicit the destination of messages. Therefore, the syntax of types is defined as \( A, B ::= a | a ⊥ | ⊥ u | 1 ˜u | (A · B) | (A ⊗ ˜u B) \), where \( a \) denotes atoms.

Multiparty compatibility. Multiparty compatibility [6, 11, 7] is a semantic notion that uses types as abstractions of the processes behaviors and simulates their execution.

In order to give a semantics to types, we introduce FIFO queues defined as \( Ψ ::= \epsilon | A · Ψ | \star · Ψ \), capturing messages that are waiting to be delivered. A message can be a proposition \( A \) or a session termination symbol \( \star \). Every pair of endpoints \( (x, y) \) has an associated queue containing messages in transit from endpoint \( x \) to endpoint \( y \). A queue environment \( σ \) is a mapping from pairs of endpoints to queues: \( σ : (x, y) \mapsto Ψ \). \( σ_{\epsilon} \) denotes the queue environment with only empty queues while \( σ[(x, y) \mapsto Ψ] \) denotes a new environment where the entry for \( (x, y) \) has been updated to \( Ψ \).

We define the type-context semantics for context \( ∆ = x_1 : B_1, \ldots, x_k : B_k \) given environment \( σ \) as the minimum relation on \( ∆ \cdot σ \) following rules in Fig. 1. Let \( α_1, \ldots, α_n \) denote a path for context \( ∆ \) if \( ∆ \cdot σ \xrightarrow{α_1} Δ_1 \cdot σ_1 \xrightarrow{α_2} \ldots \xrightarrow{α_n} Δ_n \cdot σ_n \). This path is maximal if there is no
\[
x : a^\perp, y : a \bullet \sigma_e
\]
\[
x : 1^\perp \bullet \sigma_e[(y, x) \mapsto *] \frac{x + y}{\Rightarrow} \emptyset \bullet \sigma_e
\]
\[
\Delta, x : 1^\perp \bullet \sigma_e[(x, y) \mapsto \Psi] \frac{y \downarrow x}{\Rightarrow} \emptyset \bullet \sigma_e
\]
\[
\Delta, x : A \Rightarrow^\emptyset B \bullet \sigma_e[(x, y) \mapsto \Psi] \frac{y \downarrow x}{\Rightarrow} \emptyset \bullet \sigma_e
\]
\[
\Delta, x : A \otimes^\emptyset B \bullet \sigma_e[\{(y, x) \mapsto A, \{x, y\}\}] \frac{x \downarrow y}{\Rightarrow} \emptyset \bullet \sigma_e
\]
\[
\Delta, x : \emptyset \Rightarrow A \bullet \sigma_e[\{(y, x) \mapsto \Psi\}] \frac{x \downarrow y}{\Rightarrow} \emptyset \bullet \sigma_e
\]

**Figure 1:** Type-Context Semantics

\[
P \vdash \Gamma, [\Psi][x : y : a] A \vdash B \frac{\forall x : \emptyset \bullet \sigma_e[(y, x) \mapsto *]}{\Gamma, [\Psi][x : a] \vdash B}
\]
\[
P \vdash \Gamma, [\Psi][x : y : A] A \vdash B \frac{\forall x : \emptyset \bullet \sigma_e[(y, x) \mapsto *]}{\Gamma, [\Psi][x : A] \vdash B}
\]
\[
P \vdash \Gamma, [\Psi][x : y : A] A \vdash B \frac{\forall x : \emptyset \bullet \sigma_e[(y, x) \mapsto *]}{\Gamma, [\Psi][x : A] \vdash B}
\]

**Figure 2:** Proof System for Forwarders

\[\Delta_{n+1}, \sigma_{n+1}, \alpha_{n+1} \text{ such that } \Delta_n \bullet \sigma_n \nrightarrow \Delta_{n+1} \bullet \sigma_{n+1} \text{ and it is live if } \Delta_n = \emptyset \text{ and } \sigma_n = \sigma_e, \text{ meaning that it progresses without reaching an error.}
\]

**Definition 1.** A context \( \Delta \) is multiparty compatible if all maximal paths \( \alpha_1, \ldots, \alpha_n \) for \( \Delta \) are live and such that, if \( \alpha_p = y \otimes x \mid A, \{A_i\}_i \), then \( x : A^{\perp}, \{y_i : A_i^{\perp}\}_i \) is also multiparty compatible.

**Forwarders.** To capture multiparty compatibility, forwarders are designed as a subclass of derivations in classical linear logic that satisfies three conditions: i) anything received must be forwarded, ii) anything that is forwarded must have been previously received, and iii) the order of messages between two endpoints must be preserved.

In order to enforce these properties, the logic of forwarders is defined on judgements of the form \( P \vdash \Gamma \) where \( P \) is a process and \( \Gamma := \emptyset | \Gamma, [\Psi][x : B] \) is an extended type context which associates to each endpoint \( x \) a priority queue \( [\Psi] \) consisting of any yet to process messages originating from \( x \).

Formally, \( [\Psi] := e | [u^*] [\Psi] | [u^y : A][\Psi] \) where \( [u^y : A] \) expresses that a new \( y \) of type \( A \) has been received and will need to be forwarded later to endpoint \( u \) and, similarly, \( [u^*] \) indicates that a request for closing a session has been received and must be forwarded to \( u \).

Forwarders behave asynchronously: the order of messages needing to be forwarded to independent endpoints is irrelevant unless they originate at the same \( x \). This follows the idea of having a queue for every ordered pair of endpoints in the type-context semantics above.

**Definition 2.** A process \( P \) is a forwarder for \( \Delta \) if the judgement \( P \vdash \Delta \) is derivable using the rules reported in Fig. 2.

**Correspondence.** We can conclude this abstract with the statement of the theorem that forwarders characterize multiparty compatibility.

**Theorem 3.** \( \Delta \) is multiparty compatible iff there exists a forwarder for \( \Delta \), i.e., a process \( P \) such that \( P \vdash \Delta \).

The proof, as well as more details on forwarders including additives for branching, exponentials to model servers and clients, and internal cut-elimination for forwarder composition, and applications to multiparty communication as multi-cut elimination, can be found in [4].
References


