Coherence for Asynchronous Buffered MPST: A Mechanized Metatheory

Sam Hart Berman

Abstract

Standard MPST uses coherence and projectability on global types to justify local safety. We define an operational coherence invariant $\mathsf{Coherent}(G, D)$ over local-type environment G and buffered-trace environment D, and show that this invariant is sufficient for preservation in asynchronous semantics. The operational coherence invariant yields a reusable three-way edge split that avoids per-step global re-derivation. Lean 4 mechanization confirms our results under parameterized delivery semantics.

1 Introduction

1.1 Background

Multiparty session types guarantee that distributed protocols execute without communication errors. The central technical challenge is preservation: showing that each protocol step maintains well-typedness. Binary session types use duality to ensure two-party compatibility. MPST generalizes this property by establishing global coherence and projectability conditions.

To address preservation, the MPST field has developed projection techniques that take a global choreography to local session types. A global type specifies the full protocol, then projection extracts the local view for each participant. Safety means that local executions remain consistent with their projected types.

Traditionally, MPST coherence and projectability conditions are formulated at the global-type level (Honda, Yoshida, and Carbone, 2008; Honda et al., 2016; Castagna et al., 2012), with later work refining projection criteria (Majumdar et al., 2021).

In 2008, Honda et al. introduced asynchronous buffered MPST. Their preservation proof was organized around global typing and projection consistency after transitions. Many later presentations keep this global-first organization (Honda et al., 2016; Scalas and Yoshida, 2018), while mechanized developments continue to refine proof organization and robustness (Tirore et al., 2025).

Prior MPST literature formulates coherence at the global-type level through projectability and well-formedness conditions. This paper refines coherence at the operational level by using $\mathsf{Coherent}(G, D)$ over local environments and buffered traces. Our operational formulation makes preservation local and compositional and removes the long-standing per-step global re-derivation bottleneck. We confirm this result via Lean 4 mechanization. Terminology note: throughout this paper, "coherence" denotes the operational predicate $\mathsf{Coherent}(G, D)$, not the classical global-type projectability notion. Section 2 positions this usage explicitly relative to prior MPST coherence traditions.

1.2 Contribution

Projection defines a quotient, it erases global structure while preserving safety-relevant information. The question is: what exact invariant survives this erasure? We isolate an operational form of coherence, $\mathsf{Coherent}(G, D)$, as the invariant required by our preservation proofs. Binary session types use duality for two-party compatibility. This formulation instantiates n-ary compatibility as receiver-buffer alignment at each active edge (Carbone et al., 2015).

$\mathsf{Coherent}(G, D)$ captures exactly what projection preserves for this proof architecture. It is neither too strong nor too weak, and this exactness enables compositionality. Local proofs on individual edges combine to yield global preservation. No global re-derivation is required.

DRAFT

The core contribution is one operational kernel with two theorem-level lifts: typed-step preservation and subtype replacement. The shared kernel is edge-local checking with active-edge lifting. For typed transitions, this kernel uses a three-way edge split. For subtype replacement, it relies on `consume_mono` under receive-compatibility.

The local-to-global shape follows standard invariant methods from program logics (Reynolds, 2002; O'Hearn, 2007), while aligning with process-calculus decomposition tradition (Plotkin, 1981; Milner, 1999) and Lyapunov-style invariant-function reasoning in dynamical systems (Lyapunov, 1892; LaSalle, 1960).

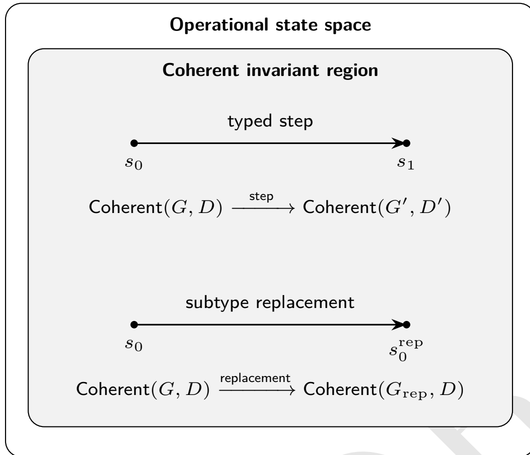

Figure 1.1: Invariant-geometric view of preservation and subtype evolution.

1.3 Scope

This paper has two central claims. The first is Coherence preservation through a three-way split that applies uniformly across transitions. The second is session evolution under subtype replacement via `consume_mono`, which we lift to global Coherence and progress-condition preservation under compatibility premises.

This paper is the first in a three-part series on asynchronous buffered semantics with DeliveryModel parameterization, and precisely defines the invariant kernel. *Computable Dynamics for Asynchronous MPST* develops quantitative bounds and decision procedures on that kernel. *Harmony from Coherence in Asynchronous MPST* proves reconfiguration commutation and envelope exactness.

More precisely, our contributions in this paper are as follows:

1. A mechanized operational Coherence architecture with a reusable three-way edge split and typed-step wrappers that lift edge-local lemmas to preservation and progress interfaces.
2. A message-alignment proof kernel based on `Consume`, with lemmas `consume_append` and `consume_cons`.
3. A subtype-replacement theorem stack from `consume_mono` that preserves edge Coherence, global Coherence, and progress-side conditions under receive-compatibility premises.
4. A typed effect bridge connecting protocol semantics to runtime execution, parametric in delivery model and premises.

Main text proofs are proof sketches; full machine-checked derivations are provided by the Lean anchors indexed in Appendix E. Proof-sketch template used throughout the series: state proof mode (compositional/induction/coinduction/witness construction), give one concrete intermediate transformation, then conclude with the claim interface.

DRAFT

Symbol	Meaning
$\mathsf{H}_{\mathrm{byz}}$	Byzantine characterization bundle
$\mathsf{Obs}^{\mathrm{byz}}_{\mathrm{safe}}$	Byzantine safety-visible observation
$\mathsf{Eq}^{\mathrm{byz}}_{\mathrm{safe}}$	Byzantine safety-visible equality
$\mathcal{E}_{\mathrm{byz}}$	Byzantine determinism-envelope
ByzChar	Byzantine characterization formula
ByzSafe	Byzantine safety predicate
ByzIfaceWF	Byzantine interface well-formedness
Coherent	global active-edge compatibility
EdgeCoherent	edge-local compatibility check
EdgeShares	edge-to-endpoint incidence predicate
Consume	trace-to-type alignment function
GEnv	endpoint-to-local-type environment
DEnv	edge-to-buffered-trace environment
(G, D)	configuration pair of local-type and buffered-trace environments
s_i	illustrative configuration states in traces/figures
q	quotient projection to symmetry classes
$\sim$	coherence-preserving symmetry/equivalence relation
ActiveEdge	endpoint-presence guard
RecvCompatible	receive-side replacement compatibility
DeliveryModel	parametric delivery interface
N_s	explored state count in finite checker objects
N_e	explored transition-edge count in finite checker objects
$L_{\max}$	maximum active-edge buffered trace length
$W_{\max}$	maximum size of checked witness/certificate object

Table 1.1: Notation used in this paper.

2 Model and Semantics

Configurations are represented by endpoint and edge environments. GEnv maps endpoints to local types, DEnv maps directed edges to buffered type traces, and `Buffers` maps directed edges to runtime payload queues.

Assumption	Status
asynchronous buffered semantics	required
active-edge quantification	required
fair scheduling profile	assumed where stated
DeliveryModelparam.	required
crash-stop and Byzantine claims	interface claim only

Table 2.1: Model assumptions for exact statements in this paper.

Assumption Block 2.1. Core Model Premises.
Core claims in this paper assume asynchronous buffered semantics, active-edge quantification, and DeliveryModelparameterization. Fair scheduling assumptions apply where stated.

Assumption-block inheritance policy. Assumption blocks do not inherit implicitly. If a theorem does not name a block, it uses Assumption Block 2.1 only.

Definition 2.1. Active Edge.
$\mathsf{ActiveEdge}(G, e)$ holds when both endpoints of edge e are present in G.

Definition 2.2. Edge Coherence.
$\mathsf{EdgeCoherent}(G, D, e)$ holds when the receiver side can consume the buffered trace on edge e under the local type in G.

Definition 2.3. Coherence.
$\mathsf{Coherent}(G, D)$ holds when every active edge is edge coherent.

Definition 2.4. Well-Typed Configuration and Step.
Write $\Gamma \vdash (G, D)\ \mathsf{wf}$ for configuration typing and $\Gamma \vdash (G, D) \to (G', D')$ for one-step typing in the operational fragment used by this paper.

Role split. Well-typedness constrains rule formation and environment-domain consistency. Coherence constrains edge-local message and type compatibility.

Definition 2.5. consumeOne.
For source role r, value type T, and local type L:

DRAFT

write $\mathsf{consumeOne}_{r,T}(L) := \mathsf{consumeOne}(r, T, L)$. Then

$$\mathsf{consumeOne}_{r,T}(\mathsf{recv}(r, T, L')) = \mathsf{some}(L'), \quad \mathsf{consumeOne}_{r,T}(L) = \mathsf{none} \quad \text{otherwise.}$$

Here NonRecvHeads abbreviates $\{\mathsf{send}, \mathsf{select}, \mathsf{branch}, \mathsf{end}, \mathsf{var}\}$. "Otherwise" means $L \in \mathsf{NonRecvHeads}$ or $L = \mu L_0$ at this layer. Head checks are therefore constructor-local. Recursive unfolding is handled by the local-type normalization layer before this head check.

Definition 2.6. Consume.
For source role r, receiver local type L, and buffered trace ts:

```
Consume r L [] :=some L
Consume r L (T :: ts) :=
  match consumeOne r T L with
  | some L' => Consume r L' ts
  | none => none
```

The recursion is structural on trace length.

Definition 2.7. Guarded Recursion Premise.
A recursive local type satisfies GuardedMu when every bound-variable occurrence in a μ-body appears under a communication constructor. This excludes unguarded heads that can unfold forever without consuming input.

Termination measure. Define

$$\mu_{\mathsf{Consume}}(L, ts) := |ts|. \tag{2.1}$$

If a recursive call is taken, the argument changes from $(L, T :: ts)$ to (L', ts), so μ_{Consume} strictly decreases by one. Hence recursion is well-founded on natural numbers.

Recursive local types. For μ-types, unfolding is handled in the normalization layer and does not add recursive depth to Consume itself, because Consume recursion is indexed only by trace length. Under GuardedMu, normalization cannot generate infinite head-unfolding chains in the well-typed fragment used here.

Mechanized anchors for this block are `consume_one_depth_lt`, `consume_length_le_depth`, `LocalTypeR.guarded_full_unfold_not_var`, and `mu_height_substitute_guarded`.

Definition 2.8. Byzantine Interface Well-Formedness.
$\mathsf{ByzIfaceWF}(G, D)$ is the interface predicate used for Byzantine-facing statements in this paper, and is mechanized as a definitional alias of $\mathsf{Coherent}(G, D)$.

Delivery behavior is parameterized by DeliveryModel. This gives one theorem shape across FIFO, causal, and lossy instances. The interface laws and instance examples appear in Appendix C.

$$\mathsf{Coherent}(G, D) \ := \ \forall e,\ \mathsf{ActiveEdge}(G, e) \rightarrow \mathsf{EdgeCoherent}(G, D, e). \tag{2.2}$$

Edges incident to absent endpoints are excluded by ActiveEdge.

2.1 Operational Coherence

In this paper, "coherence" means the operational invariant $\mathsf{Coherent}(G, D)$ from Equation 2.2: every active edge must satisfy EdgeCoherent.

This is aligned with, but distinct from, classical MPST coherence/projectability on global types (Honda et al., 2008; Honda et al., 2016; Castagna et al., 2012; Majumdar et al., 2021) and logical n-ary coherence accounts (Carbone et al., 2015; Carbone et al., 2016; Carbone et al., 2017). The present claim is operational and proof-architectural, not a denotational identification of models.

The compatibility intuition follows coherence-space/session-logic lineages (Girard, 1987; Caires and Pfenning, 2010; Wadler, 2012) and the communicating-automata compatibility tradition (Gay and Hole, 2005; Deniélou and Yoshida, 2012; Brand and Zafiropulo, 1983): local compatibility obligations aggregate into one global invariant used by preservation.

Coherence-space	Operational in This Paper
token	active-edge obligation for directed edge e
coherence relation	$\mathsf{EdgeCoherent}(G, D, e)$
clique	$\mathsf{Coherent}(G, D)$ over all active edges
preservation of compatibility	one-step preservation (Thm. 4.1)

Table 2.2: Compatibility correspondence used as operational intuition (not a formal embedding claim).

DRAFT

3 Worked Example

We illustrate coherence preservation through a three-role protocol that exercises the key proof obligations. These include active-edge checks, buffer compatibility via $\mathsf{Consume}$, and the three-way edge split for typed steps. The configuration includes a non-empty buffer to demonstrate how in-flight messages interact with local-type advancement.

Running example. Roles are `C` (client), `P` (pool), and `M` (monitor), with global interaction:

```
C → P : Request(n)
P → C : Grant(k)
C → M : Report(k)
M → P : Confirm
P → C : Token(t)
```

Fix local continuations:

```
L_C :=recv P Grant(k);
      send M Report(k);
      recv P Token(t); end
L_P :=recv M Confirm;
      send C Token(t); end
L_M :=recv C Report(k);
      send P Confirm; end
```

and in-flight configuration (G_0, D_0):

```
G_0(C) = L_C
G_0(P) = L_P
G_0(M) = L_M
D_0(P,C) = [Grant(k)]
D_0(e) = [] for e != (P,C)
```

Assume typing judgment $\Gamma \vdash (G_0, D_0)\ \mathsf{wf}$.
The active-edge obligation is:

$$\forall e,\ \mathsf{ActiveEdge}(G_0, e) \to \mathsf{EdgeCoherent}(G_0, D_0, e). \tag{3.1}$$

The example is used via the following derived rule instances.

$$\frac{\Gamma \vdash (G_0, D_0)\ \mathsf{wf} \quad D_0(P, C) {=} [\mathrm{Grant}(k)]}{(G_0, D_0) {\to} (G_1, D_1) \wedge \mathsf{Coherent}(G_1, D_1)} \tag{3.2}$$

$$\frac{\Gamma \vdash (G_1, D_1)\ \mathsf{wf} \quad \texttt{consume_append}}{(G_1, D_1) {\to} (G_2, D_2) \wedge \mathsf{Coherent}(G_2, D_2)} \tag{3.3}$$

$$\frac{\mathsf{Coherent}(G_0, D_0) \quad \mathsf{RecvCompatible}(G_0, G_0')}{\mathsf{Coherent}(G_0', D_0)} \tag{3.4}$$

For the step edge $u = (C, M)$ with stepped endpoint ep_{step}, the case split used later is: updated $(e = u)$, shared-endpoint $(e \neq u \wedge \mathsf{EdgeShares}(e, ep_{\mathrm{step}}))$, and unrelated $(e \neq u \wedge \neg\mathsf{EdgeShares}(e, ep_{\mathrm{step}}))$.

4 Coherence Preservation Architecture

Running-example thread. Section 3 instantiates this theorem on the baseline C/P/M trace and its one-step send-recv transitions.

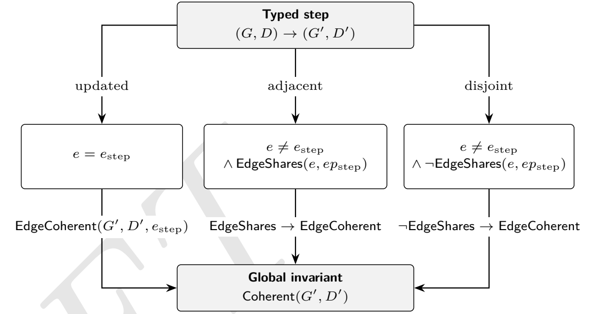

Figure 4.1: Three-way edge split used by the preservation proof. Top node is the typed step $(G, D) \to (G', D')$. Middle nodes are the exhaustive edge classes (updated, adjacent/shared-endpoint, disjoint). Bottom node is the global post-step invariant. Arrow labels state the obligation discharged per class.

Theorem 4.1. Coherence Preservation. *Under Assumption Block 2.1, for all environments G, D, G', D', if $(G, D) \to (G', D')$ is a well-typed step in the send and recv and select and branch fragment, then*

$$\mathsf{Coherent}(G, D) \implies \mathsf{Coherent}(G', D'). \tag{4.1}$$

This claim is exact for one-step preservation in the stated core rule fragment.

Premise roles. In Theorem 4.1, well-typedness selects the applicable operational rule and preserves structural side conditions, while Coherence provides the edge obligations transported by the three-way split.

Running-example instantiation. In Section 3, each checked transition in the baseline protocol is one of the send/recv/select/branch steps covered by this theorem.

Lemma 4.2. Three-Way Edge-Split Exhaustiveness.

DRAFT

Fix stepped edge u and stepped endpoint ep_{step}. For any edge e, exactly one of the following holds:

$$e = u,$$
$$e \neq u \wedge \mathsf{EdgeShares}(e, ep_{\text{step}}),$$
$$e \neq u \wedge \neg\mathsf{EdgeShares}(e, ep_{\text{step}}).$$

Proof sketch. Case-split first on $e = u$. If false, split on the Boolean incidence predicate $\mathsf{EdgeShares}(e, ep_{\text{step}})$. This yields the two remaining cases and excludes overlap by propositional contradiction. Graph-theoretically, this is the partition of all non-updated edges into those incident to the stepped endpoint and those not incident to it. The mechanized splitter is `edge_case_split`. □

Proof. Fix a well-typed one-step transition

$$(G, D) \to (G', D')$$

in the send/recv/select/branch fragment, and assume $\mathsf{Coherent}(G, D)$. To prove $\mathsf{Coherent}(G', D')$, let e be arbitrary and assume $\mathsf{ActiveEdge}(G', e)$. It suffices to show $\mathsf{EdgeCoherent}(G', D', e)$.

Apply `edge_case_split` to e with respect to the stepped edge e_{step} and stepped endpoint ep_{step}. Exactly one of the following holds:

$$e = e_{\text{step}},$$
$$e \neq e_{\text{step}} \wedge \mathsf{EdgeShares}(e, ep_{\text{step}}),$$
$$e \neq e_{\text{step}} \wedge \neg\mathsf{EdgeShares}(e, ep_{\text{step}}).$$

Case 1 (updated edge). Here $e = e_{\text{step}}$. The post-step edge obligation is discharged by the rule-local preservation theorem corresponding to the operational rule used by $(G, D) \to (G', D')$: send/recv/select/branch map to `coherent_send_preserved`, `coherent_recv_preserved`, `coherent_select_preserved`, `coherent_branch_preserved`. The message/type-alignment subgoal is reduced to $\mathsf{Consume}$ lemmas: send/select use `consume_append` for enqueue extension, recv/branch use `consume_cons` for aligned head-consumption and continuation alignment. Hence $\mathsf{EdgeCoherent}(G', D', e)$. In the running example (Section 3), this is the direct case for the currently stepped channel edge.

Case 2 (shared endpoint, not updated edge). Assume $e \neq e_{\text{step}} \wedge \mathsf{EdgeShares}(e, ep_{\text{step}})$. From pre-state coherence and active-edge premises on unchanged incident structure, we have $\mathsf{EdgeCoherent}(G, D, e)$. The shared-endpoint transport lemmas preserve receiver projection and the relevant buffered segment on e, so the same consumption witness transports to the post-state. Therefore $\mathsf{EdgeCoherent}(G', D', e)$.

Case 3 (unrelated edge). Assume $e \neq e_{\text{step}} \wedge \neg\mathsf{EdgeShares}(e, ep_{\text{step}})$. No endpoint or buffer component of e is touched by the step; frame transport gives invariance of edge-local coherence on e between (G, D) and (G', D'). Since $\mathsf{Coherent}(G, D)$, we obtain $\mathsf{EdgeCoherent}(G', D', e)$.

All cases yield $\mathsf{EdgeCoherent}(G', D', e)$ for arbitrary active e, hence

$$\forall e,\ \mathsf{ActiveEdge}(G', e) \to \mathsf{EdgeCoherent}(G', D', e),$$

which is $\mathsf{Coherent}(G', D')$. □

The split operator is implemented by `edge_case_split`. Rule-level preservation lemmas are grouped by send, recv, select, and branch cases. Table E.1 gives the anchor-to-file mapping, and section B records theorem-to-lemma chains.

The architecture is quotient-first. The concrete step relation induces dynamics on equivalence classes once coherence-preserving symmetries are quotiented out.

The quotienting step should be read as symmetry reduction. Distinct interleavings that are equivalent under the symmetry relation are identified as one observable class. Formally, the induced square is:

Definition 4.3. Observational Quotient Relation.
For coherent configurations X, Y, write $X \sim Y$ iff their runtime realizations are equivalent under the configuration-equivalence/effect-bisimulation bridge used in this paper (equivalently, effect-bisimilar modulo silent steps at the observational interface).

Relation note. This paper writes the quotient as $\sim$; the same bridge-level relation is named `ConfigEquiv` in Paper 3. The quotient collapses silent internal monitor steps and symmetry-equivalent reorderings that preserve safety-visible observations. Canonical alias across the series: on coherent configurations, $(G_1, D_1) \sim (G_2, D_2)$ iff `ConfigEquiv`$((G_1, D_1), (G_2, D_2))$ at the observational boundary defined in Paper 3 (Configuration

DRAFT

Equivalence definition). Scope is the same in both papers: coherent/admitted states only.

$$q(\mathsf{step}(G, D)) \;=\; \mathsf{step}_{/\sim}(q(G, D)). \tag{4.2}$$

Here q is the quotient projection map and $\sim$ is the coherence-preserving equivalence relation used by the bridge layer. Equation 4.2 states the commuting condition used across these three papers.

Lemma 4.4. Step Congruence Under $\sim$. *For coherent pre-states (G_1, D_1) and (G_2, D_2), if*

$$(G_1, D_1) \sim (G_2, D_2)$$

and

$$(G_1, D_1) \to (G_1', D_1'),$$

then there exists (G_2', D_2') such that

$$(G_2, D_2) \to (G_2', D_2')$$

and

$$(G_1', D_1') \sim (G_2', D_2').$$

Proof sketch. Use the configuration-equivalence and effect-bisimulation bridge (`config_equiv_iff_effect_bisim_silent` and `effect_bisim_implies_observational_equivalence`) to transport one step across $\sim$-equivalent pre-states. Then apply Theorem 4.1 to keep post-states in the coherent domain of the quotient relation. This argument is compositional at the one-step level: first transport the operational step across the quotient, then compose with preservation on the target side. No separate induction is needed because the statement is single-step. □

The commuting equation above is therefore read as preservation plus Lemma 4.4, which makes $\mathsf{step}_{/\sim}$ well-defined on quotient classes.

The case partition used by `edge_case_split` is: *Case partition for checked edge.*

Here $S(e) := \mathsf{EdgeShares}(e, ep_{\text{step}})$.

$$\begin{aligned} \mathsf{updated}: &\;\; e = u; \\ \mathsf{shared}: &\;\; e \neq u \wedge S(e); \\ \mathsf{unrelated}: &\;\; e \neq u \wedge \neg S(e). \end{aligned} \tag{4.3}$$

The three branches are discharged by updated-edge preservation, shared-endpoint transport, and unrelated-edge frame transport, respectively.

5 Message-Type Alignment via Consume

Consume is the proof kernel for message-type alignment. It removes most rule-specific structural complexity from preservation proofs.

Lemma 5.1. consume_append. *Consumption over concatenated traces factors through sequential consumption.*

Lemma 5.2. consume_cons. *Head consumption reduces to one-step alignment plus recursive continuation alignment.*

Proof sketch. The library is centered on consumeOne and structural recursion on trace shape.

1. `consume_append`: induction on trace prefix ts; base case $ts = []$ is immediate by simplification ($L' = L_r$); step case peels one consumeOne transition and applies induction hypothesis to the residual local type.
2. `consume_cons`: unfold one Consume step on head T; case split on $\mathsf{consumeOne}(\mathit{from}, T, L_r)$; resulting equation is definitional in both branches.
3. Preservation integration: send/select proofs use `consume_append` to justify extending buffered traces; recv/branch proofs use `consume_cons` to justify consuming the head message and continuing.

This isolates alignment complexity into two reusable lemmas rather than duplicating ad-hoc trace reasoning in each operational rule. □

For the worked example, `consume_cons` discharges the `Grant` receive step at `C` by reducing the obligation to continuation coherence after one aligned message.

Complexity note (checker vs runtime). Under constant-time constructor tests in consumeOne, one Consumecheck on edge trace ts_e costs $O(|ts_e|)$. Therefore one full active-edge Coherence *checker* pass costs

$$O\Big(\sum_{e \in E_{\text{active}}} |ts_e|\Big) = O(N_e \cdot L_{\max}),$$

where $N_e := |E_{\text{active}}|$ and $L_{\max} := \max_{e \in E_{\text{active}}} |ts_e|$. Runtime *system-step* cost

DRAFT

is separate: this bound is for validation/checking, not for the operational transition relation itself.

6 Session Evolution via Subtyping

Running-example thread. Section 3 gives the concrete endpoint state that is refined by this replacement theorem.

Session evolution replaces one endpoint local type with a compatible refinement and asks whether coherence is preserved.

This replacement argument follows a standard commutation intuition from trace-equivalence and commuting-conversion lines. The standard point is that local reorderings or local refinements should preserve observable behavior when compatibility conditions hold. What is new here is an operational criterion for asynchronous subtype replacement that is stated directly through `consume_mono` and discharged with edge-local coherence obligations.

Assumption Block 6.1. Replacement Compatibility.
The subtype-evolution theorem assumes endpoint replacement at *ep* satisfies typing-domain preservation, receive-side monotonicity side conditions, and environment consistency for unaffected endpoints.

Theorem 6.1. Session Evolution via `consume_mono`.
For all endpoints ep and environments G, D, G_{rep}*, if Assumption Block 6.1 holds for replacement at ep, then*

$$\mathsf{Coherent}(G, D) \implies \mathsf{Coherent}(G_{\text{rep}}, D). \tag{6.1}$$

This claim is exact for type replacement steps covered by Assumption Block 6.1.

Running-example instantiation. In Section 3, replacing one receiver continuation by a compatible subtype is an instance of this theorem.

Proof sketch. The replacement theorem is proved by a monotonicity-to-global-lift chain.

1. Local monotonicity: establish receive-compatibility ($\mathsf{RecvCompatible}$) between old and replacement local types; apply `consume_mono` to show every successful old consumption remains successful after replacement.
2. Edge lift: use `edge_coherent_type_replacement` on edges targeting the replaced endpoint; for non-target edges, transport coherence by environment agreement.
3. Global lift: combine edge cases to derive `coherent_type_replacement`.
4. Progress compatibility: use liveness-side lemmas (`progress_conditions_type_replacement`) to preserve operational side conditions.

In the running example, this appears as refining the continuation after the first request/grant exchange while preserving all edge checks.

Hence compatible subtype replacement preserves Coherence without re-running global derivations. □

Core lemmas are in the subtype-replacement core and liveness layers. The subtype-replacement commutation principle and proof-lift chain are:

$$\mathsf{Coherent}(G, D) \ \wedge\ \mathsf{Compat}_{ep}(G, G_{\text{rep}}) \implies \mathsf{Coherent}(G_{\text{rep}}, D). \tag{6.2}$$

$$\texttt{consume_mono} \implies \text{edge lift} \implies \text{global lift}. \tag{6.3}$$

with compatibility discharged by `consume_mono` and replacement side conditions.

7 Effect-Typed Bridge

`EffectSpec.handlerType` records typed effect obligations at the VM boundary. The VM instruction layer used in this paper has **`send`**, **`recv`**, **`select`**, **`branch`**, and **`invoke`** forms. `WellTypedInstr` maps each instruction form to a session-side obligation. Send and select forms use sender-side continuation obligations. Receive and branch forms use $\mathsf{Consume}$-side alignment obligations. Invoke form lifts handler obligations to instruction fragments. Write this mapping as

$$\begin{aligned}
\mathsf{WTObl}(\texttt{send}) &= \mathsf{SendCont}, \\
\mathsf{WTObl}(\texttt{recv}) &= \mathsf{RecvConsumeCont}, \\
\mathsf{WTObl}(\texttt{select}) &= \mathsf{SelectBranchCont}, \\
\mathsf{WTObl}(\texttt{branch}) &= \mathsf{BranchCaseCont}, \\
\mathsf{WTObl}(\texttt{invoke}) &= \mathsf{InvokeLift}.
\end{aligned} \tag{7.1}$$

where each right-hand side names the corresponding session obligation family in `WellTypedInstr`.

DRAFT

Effect-bisimulation primer. Two configurations are effect-bisimilar when they match on the same observable effect trace up to silent internal steps. In this paper, effect-bisimulation is the bridge quotient because bridge soundness is an observational claim, and silent monitor-internal variation should not change that claim.

Instruction Form	Session-Side Typing Obligation
`send`	continuation typing with enqueue witness
`recv`	Consume-alignment witness with continuation typing
`select`, `branch`	label agreement with continuation typing
`invoke`	handler-obligation lift to typed VM fragment

Table 7.1: VM instruction-layer summary used by the bridge theorem.

Proposition 7.1. Bridge Soundness.
Under Assumption Block 2.1, for any selected handler h, if choreography obligations for h are discharged, `EffectSpec.handlerType` holds for h, `monitor_ sound` and `unified_ monitor_ preserves` hold for the active monitor, and the configuration-equivalence/effect-bisimulation bridge assumptions hold, then VM-side typing obligations generated from h are satisfied in the corresponding runtime instruction layer and transport to the observational interface.

Proof sketch. The bridge composition has three steps.

1. Build `VMBridgePremises` from `monitor_sound`, `unified_monitor_preserves`, and the `WellTypedInstr.wt_invoke` witness.
2. Lift choreography obligations to instruction fragments and VM-step typing through `handler_obligation_local`, `handler_fragment_typing_of_premises`, and `handler_vm_step_typing`.
3. Transport typed VM steps to observations via `vm_bridge_soundness_composed`, `config_equiv_iff_effect_bisim_silent`, and `effect_bisim_implies_observational_equivalence`.

Equivalently, with $\mathsf{BridgePrem}(h)$ abbreviating the listed premises,

$$\mathsf{BridgePrem}(h) \Longrightarrow \mathsf{InstrTyped}(h) \land \mathsf{ObsTransport}(h).$$

Therefore runtime typing obligations and observational transport both hold under the stated premises. □

Assumption Block 7.1. Byzantine Interface Premises.
The interface statements in this section assume shared notation for $\mathsf{H}_{\mathrm{byz}}$, $\mathsf{Obs}^{\mathrm{byz}}_{\mathrm{safe}}$, $\mathsf{Eq}^{\mathrm{byz}}_{\mathrm{safe}}$, and $\mathcal{E}_{\mathrm{byz}}$, plus theorem-pack capability naming consistency between abstract and runtime layers.

Theorem 7.2. Byzantine Interface Well-Formedness (BZ-1).
Under Assumption Block 7.1, Byzantine safety-track statements in this paper are interpreted on configurations (G, D) satisfying $\mathsf{ByzIfaceWF}(G, D)$ (definitional alias of $\mathsf{Coherent}(G, D)$). For any active edge, this interface yields the same sender-existence and Consume *obligations as the Coherence layer.*

Corollary 7.3. Dropped-Assumption Witness Interface (BZ-1.1).
Under Assumption Block 7.1, let $\mathcal{A}$ be any assumption class named in $\mathsf{H}_{\mathrm{byz}}$. If a later construction provides a witness violating ByzSafe*when $\mathcal{A}$ is dropped, then that witness can be expressed against the same active-edge Coherence interface used in this paper.*

Proposition 7.4. Capability-Gated VM Interface (BZ-1.2).
Under Assumption Block 7.1, if a runtime profile includes Byzantine characterization and envelope-adherence capability evidence, then safety-visible runtime obligations are interpreted through $\mathsf{Eq}^{\mathrm{byz}}_{\mathrm{safe}}$modulo $\mathcal{E}_{\mathrm{byz}}$, and profile admission is blocked when required evidence is absent.

8 Related Work

Classical MPST establishes global coherence and projectability as conditions on global types and projection (Honda et al., 2008, Honda et al., 2016, Castagna et al., 2012, and Majumdar et al., 2021). Logical lines interpret multiparty compatibility as n-ary duality and proof compatibility (Carbone et al., 2015, Carbone et al., 2016, and Carbone et

DRAFT

al., 2017). Data certification extensions remain in the same global-projection discipline (Toninho and Yoshida, 2017).

This paper refines the established notion of coherence in MPST with an operational invariant $\mathsf{Coherent}(G, D)$ over local environments and buffered traces. The novelty is an asynchronous preservation architecture that discharges obligations edge-locally and avoids per-step global rederivation.

Local-first and rely/guarantee lines motivate alternatives to global-first checks (Scalas and Yoshida, 2018). Program-logical lines such as Actris target implementation proofs over protocol resources at a different verification layer (Hinrichsen et al., 2020). Event-structure and partial-order lines provide complementary macro semantics for concurrency (Castellan et al., 2023).

Compared with these lines, this paper contributes a reusable proof kernel that combines a three-way edge split, Consume-based message alignment, subtype-replacement monotonicity, and delivery-parametric preservation under explicit premises. It also gives a premise-conditional VM bridge chain that composes monitor contracts with effect-bisimulation transport.

Prior limitation	**Contribution**
preservation tied to global re-derivation	reusable edge-local skeleton
weak theory/execution boundary	explicit effect-typed bridge
FIFO-centric semantics	one theorem under DeliveryModel
fragmented proof structure	split kernel + Consumelemmas

Table 8.1: Comparison with prior work.

9 Limitations and Scope

This paper proves preservation and progress-side condition preservation for the typed asynchronous core rule family through the operational Coherence architecture. We also prove subtype replacement stability for edge Coherence, global Coherence, and progress-side conditions when replacement preserves typing domains, satisfies receive-side monotonicity, and keeps unaffected endpoints environment-consistent. VM bridge claims are conditional on explicit monitor and quotient-transport premises rather than unconditional runtime metatheorems.

Byzantine material in this paper is interface-layer only (Theorem 7.2 and its corollary/proposition wrappers), not a full fault-characterization envelope theorem. Quantitative, full fault-characterization, and envelope-level claims are proved in later papers within this series.

Excluded protocol classes in this paper include unguarded recursive local types, non-typed reconfiguration operators, and liveness claims under adversarial loss or Byzantine scheduling.

Operational failure mode. If active-edge Coherence premises fail, monitor-side consume/alignment checks fail on some edge and the corresponding typed step is not admitted (operationally: blocked progress or rejected transition at the typed interface). If subtype-replacement side conditions fail, replacement is treated as outside theorem scope and must be rejected by the proof-carrying admission boundary. Concrete diagnostic. In theorem-pack terms, expect an admission report that names the failing edge-level consume/alignment obligation, or a failed `canOperateUnderByzantineEnvelope` capability gate when Byzantine-side evidence is missing.

Implementation pointer. The theorem-facing interfaces map to `Protocol/Coherence/*.lean` for the core invariant and `Runtime/Proofs/VM/BridgeStrengthening.lean` with `Runtime/VM/Runtime/Monitor.lean` for VM bridge obligations.

10 Conclusion

Since Honda et al. (2008), many preservation accounts for asynchronous MPST have been organized around global re-derivation after each step. This paper provides a mechanized alternative.

The key insight is that projection from global to local types defines an erasure. The operational invariant $\mathsf{Coherent}(G, D)$ is exact for our preservation architecture. Because it is exact at the operational boundary, local proofs compose without global rederivation.

The resulting kernel is small, compositional, and extensible. It supports subtype-driven session evolution through a single monotonicity lemma. The

DRAFT

mechanization in Lean 4 confirms that the architecture scales.

Works Cited

Baier, C., and Katoen, J.-P. (2008). Principles of Model Checking. MIT Press.

Brand, D., and Zafiropulo, P. (1983). On Communicating Finite-State Machines. Journal of the ACM, 30(2), 323–342.

Caires, L., and Pfenning, F. (2010). Session Types as Intuitionistic Linear Propositions. CONCUR 2010.

Carbone, M., Lindley, S., Montesi, F., Schürmann, C., and Wadler, P. (2016). Coherence Generalises Duality: A Logical Explanation of Multiparty Session Types. CONCUR 2016.

Carbone, M., Montesi, F., Schürmann, C., and Yoshida, N. (2015). Multiparty Session Types as Coherence Proofs. CONCUR 2015.

Carbone, M., Montesi, F., Schürmann, C., and Yoshida, N. (2017). Multiparty Session Types as Coherence Proofs. Acta Informatica, 54(3), 243–269.

Castagna, G., Dezani-Ciancaglini, M., Gesbert, N., and Padovani, L. (2012). On Global Types and Multi-Party Sessions.

Castellan, S., et al. (2023). Event-structure and partial-order semantics for session-based concurrency. Journal of Logic and Algebraic Methods in Programming.

Cover, T. M., and Thomas, J. A. (2006). Elements of Information Theory (2nd ed.). Wiley.

Deniélou, P.-M., and Yoshida, N. (2012). Multiparty Session Types Meet Communicating Automata. ESOP 2012.

Gay, S. J., and Hole, M. (2005). Subtyping for Session Types in the Pi Calculus. Acta Informatica, 42(2–3), 191–225.

Girard, J.-Y. (1987). Linear Logic. Theoretical Computer Science, 50(1), 1–101.

Hinrichsen, J., et al. (2020). Actris: Session-type based reasoning in separation logic. POPL 2020.

Hoare, C. A. R. (1985). Communicating Sequential Processes. Prentice Hall.

Honda, K. (1993). Types for Dyadic Interaction. CONCUR 1993.

Honda, K., Vasconcelos, V. T., and Kubo, M. (1998). Language Primitives and Type Discipline for Structured Communication-Based Programming. ESOP 1998.

Honda, K., Yoshida, N., and Carbone, M. (2008). Multiparty Asynchronous Session Types. POPL 2008.

Honda, K., Yoshida, N., and Carbone, M. (2016). Multiparty Asynchronous Session Types. Journal of the ACM, 63(1), Article 9.

LaSalle, J. P. (1960). Some Extensions of Liapunov's Second Method. IRE Transactions on Circuit Theory, 7(4), 520–527.

Lyapunov, A. M. (1892). The General Problem of the Stability of Motion. Kharkov Mathematical Society.

Majumdar, R., Mukund, M., Stutz, F., and Zufferey, D. (2021). Generalising Projection in Asynchronous Multiparty Session Types. CONCUR 2021.

Milner, R. (1999). Communicating and Mobile Systems: The Pi-Calculus. Cambridge University Press.

Milner, R., Parrow, J., and Walker, D. (1992). A Calculus of Mobile Processes, I and II. Information and Computation, 100(1), 1–77.

O'Hearn, P. W. (2007). Resources, Concurrency, and Local Reasoning. Theoretical Computer Science, 375(1–3), 271–307.

Pierce, B. C. (2002). Types and Programming Languages. MIT Press.

Plotkin, G. D. (1981). A Structural Approach to Operational Semantics. DAIMI FN-19.

Reynolds, J. C. (2002). Separation Logic: A Logic for Shared Mutable Data Structures. LICS 2002.

Scalas, A., and Yoshida, N. (2018). Multiparty Session Types, Beyond Duality. Journal of Logical and Algebraic Methods in Programming, 97, 55–84.

Shannon, C. E. (1948). A Mathematical Theory of Communication. Bell System Technical Journal, 27(3), 379–423 and 27(4), 623–656.

Tirore, L., Bengtson, J., and Carbone, M. (2025). Mechanized MPST metatheory with subject-reduction robustness analysis. ECOOP 2025.

Toninho, B., and Yoshida, N. (2017). Certifying Data in Multiparty Session Types. Journal of

DRAFT

Logical and Algebraic Methods in Programming, 90, 61–83.

Wadler, P. (2012). Propositions as Sessions. ICFP 2012.

DRAFT

Appendix

A Syntax and Judgments

This appendix fixes the formal objects and judgment forms used in Sections 2–6.

A.1 Roles, Endpoints, and Edges

Let `Role` be the finite set of role names. An endpoint is a pair (sid, r) of session identifier and role. A directed edge is a triple (sid, r_s, r_r) with sender role r_s and receiver role r_r.

We write $\mathsf{src}(e)$ for the sender endpoint of edge e, $\mathsf{dst}(e)$ for the receiver endpoint, $\mathsf{role}(ep)$ for the role component of endpoint ep, and $\mathsf{EdgeShares}(e, ep)$ when endpoint ep is either $\mathsf{src}(e)$ or $\mathsf{dst}(e)$.

A.2 Local Types and Environments

Local types are generated by:

```
L ::= end
    | send r T L
    | recv r T L
    | select r {li : Li}i
    | branch r {li : Li}i
    | mu L
```

A typing environment G maps endpoints to local types. A delayed-trace environment D maps directed edges to buffered type traces.

A.3 Active Edges and Coherence

An edge is active exactly when both its endpoints are present in G:

```
ActiveEdge G e :=(G (src e) != none) and (G (dst
      e) != none)
```

Edge coherence is a local receive-side compatibility judgment:

```
EdgeCoherent G D e :=
  exists Lr,
    G (dst e) = some Lr and
    Consume (role (src e)) Lr (D e) != none
```

Global coherence is active-edge quantification:

```
Coherent G D :=∀e, ActiveEdge G e →
     EdgeCoherent G D e
```

B Theorem-to-Lemma Chains and Lean Anchors

This appendix records compact proof skeletons for theorem-level claims. Each step names the Lean anchors it relies on. File paths for all anchors appear in Table E.1.

B.1 Theorem 4.1: Coherence Preservation

1. Partition each checked edge into three cases via `edge_case_split`.
2. Discharge updated-edge obligations via `coherent_send_preserved`, `coherent_recv_preserved`, `coherent_select_preserved`, and `coherent_branch_preserved`.
3. Discharge alignment obligations via `consume_append` and `consume_cons`.
4. Reassemble to global active-edge quantification for $\mathsf{Coherent}$.

B.2 Theorem 6.1: Session Evolution via `consume_mono`

1. Establish receive-compatibility premise at the replacement endpoint.
2. Lift consumption success along replacement via `consume_mono`.
3. Lift edge-local replacement via `edge_coherent_type_replacement`.
4. Lift globally via `coherent_type_replacement`. Preserve progress side conditions via `progress_conditions_type_replacement`.

B.3 Proposition 7.1: Bridge Soundness

1. Build a `VMBridgePremises` bundle from `monitor_sound` and `unified_monitor_preserves`.
2. Lift handler obligations via `handler_obligation_local`, `handler_fragment_typing_of_premises`, and `handler_vm_step_typing`.
3. Compose via `vm_bridge_soundness_composed`, `config_equiv_iff_effect_bisim_silent`, and `effect_bisim_implies_observational_equivalence`.

DRAFT

B.4 Theorem BZ-1: Byzantine Interface Well-Formedness

1. Expand ByzIfaceWFto Coherentvia `byz_iface_wf_iff_coherent`.
2. On each active edge, extract sender witness and Consumesuccess via `bz1_byzantine_interface_well_formedness`.

B.5 Proposition BZ-1.2: Capability-Gated VM Interface

1. Gate runtime operation via `canOperateUnderByzantineEnvelope`.
2. Extract Byzantine envelope adherence via `vm_byzantine_envelope_adherence_of_witness`.
3. Package cross-target conformance via `byzantine_cross_target_conformance_of_witnesses`.

C Delivery Parametricity and Runtime Bridge

Preservation is parametric over a delivery interface DeliveryModelwith a model-indexed alignment function $\mathsf{Consume}_M$.

C.1 DeliveryModel Interface

For each delivery instance M, the proof uses the following laws.

Law	Required Statement
`nil`	$\mathsf{Consume}_M(r, L, []) = \mathsf{some}(L)$
`cons`	One-step head law through consumeOne for $T :: ts$
`append`	Sequential composition on concatenated traces $ts_1 + +ts_2$
`rename`	Stability under endpoint-preserving renamings

Table C.1: Delivery interface laws used by preservation proofs.

The `cons` and `append` laws are the only places where delivery discipline enters the Consume-based proof kernel.

C.2 Instance Examples

FIFO instance. Delivered traces keep send order on each edge. This instance is provided by `fifo_delivery_laws`.

Lossy instance. Delivered traces are order-preserving subsequences of sent traces on each edge. This instance is provided by `lossy_delivery_laws`.

Safety scope under message loss. Coherence-preservation claims are stated on the post-step delayed-trace environment D'. If a delivery instance drops a message, the dropped message is absent from D'. The proof checks Consumeobligations on that actual D'. This preserves the same safety statement and does not claim liveness or eventual delivery.

D Reproducibility

Reproduction workflow and expected outputs are documented in `ARTIFACT.md`. The one-command supplement check is `just artifact-check`. Pinned-commit, DOI, and Lean-statistics rows are synchronized via `bash scripts/paper_repro_rows.sh -sync papers/paper1.tex papers/paper2.tex papers/paper3.tex`. Artifact semantics note. The Lean artifact exposes an explicit trace-level strong fairness predicate (`StrongFairEnv`) refining weak fairness, and the default in-memory runtime transport implements minimal per-edge FIFO enqueue/dequeue behavior.

Artifact Field	Value
Repository	`https://github.com/hxrts/telltale`
Pinned commit	`ccddc8ea843684c4 d2d018f673c4861d0664e008`
Archival DOI	`DOI-UNSET`
Lean source statistics	1303 files, 206913 LOC, axioms: 0, unresolved proof holes (`sorry`): 0

Table D.1: Artifact identity and verification statistics for this draft snapshot.

1. Run `just artifact-check`. This runs reproducibility row checks, `just escape`, `just verify-protocols`, paper builds, and manifest generation.

DRAFT

2. Before camera-ready submission, set DOI in `papers/artifact_metadata.env` and run `just paper-repro-check-strict`.

DRAFT

E Anchor Index

Table E.1 maps paper claims to their Lean anchors and source files. Anchors marked with — are infrastructure lemmas used in proofs but not tied to a single claim. Claim entries include assumption-block tags in square brackets. Assumption-tag legend for claim cells: [A1] = Assumption Block 2.1, [A2] = Assumption Block 6.1, [A3] = Assumption Block 7.1. Functional categories in the table are: Definitions & Structures, Predicates, Theorems, and Runtime Gates. Reading guide for long names: `handler_fragment_typing_of_premises` lifts handler premises to typed VM fragments; `config_equiv_iff_effect_bisim_silent` is the bridge equivalence law; `progress_conditions_type_replacement` preserves liveness-side side conditions under subtype replacement. Name-pattern note. Most anchors use `subject_property_of_premises` or `subject_preserves_property`; mentally read left-to-right as "object, then claim, then side-condition context."

DRAFT

Anchor	Path (relative to lean/)	Sec.	Claims
Definitions & Structures			
Consume	Protocol/Coherence/Consume.lean	§5	—
EdgeCoherent	Protocol/Coherence/EdgeCoherenceCore.lean	§2	—
Coherent	Protocol/Coherence/EdgeCoherenceCore.lean	§2	—
VMBridgePremises	Runtime/Proofs/VM/BridgeStrengthening.lean	§7	—
ByzIfaceWF	Runtime/Proofs/Adapters/Distributed/ EnvelopeTheorems.lean	§7	Thm. 7.2 [A3]
WellTypedInstr	Runtime/VM/Runtime/Monitor.lean	§7	Prop. 7.1 [A1]
Predicates			
monitor_sound	Runtime/VM/Runtime/Monitor.lean	§7	Prop. 7.1 [A1]
unified_monitor_preserves	Runtime/VM/Runtime/Monitor.lean	§7	Prop. 7.1 [A1]
Theorems			
consume_append	Protocol/Coherence/Consume.lean	§5	Thm. 4.1 [A1]
consume_cons	Protocol/Coherence/Consume.lean	§5	Thm. 4.1 [A1]
consume_one_depth_lt	Protocol/Coherence/Consume.lean	§2	Consume well-foundedness
consume_length_le_depth	Protocol/Coherence/Consume.lean	§2	Consume well-foundedness
LocalTypeR. guarded_full_unfold_not_var	SessionTypes/LocalTypeR/WellFormedness.lean	§2	Guarded recursion premise
mu_height_substitute_guarded	SessionTypes/LocalTypeR/Substitution.lean	§2	Guarded recursion premise
consume_mono	Protocol/Coherence/ SubtypeReplacementCore.lean	§6	Thm. 6.1 [A1, A2]
edge_case_split	Protocol/Coherence/Unified.lean	§4	Thm. 4.1 [A1]
coherent_send_preserved	Protocol/Coherence/Preservation.lean	§4	Thm. 4.1 [A1]
coherent_recv_preserved	Protocol/Coherence/PreservationRecv.lean	§4	Thm. 4.1 [A1]
coherent_select_preserved	Protocol/Coherence/SelectPreservation.lean	§4	Thm. 4.1 [A1]
coherent_branch_preserved	Protocol/Coherence/SelectPreservation.lean	§4	Thm. 4.1 [A1]
edge_coherent_type_replacement	Protocol/Coherence/ SubtypeReplacementCore.lean	§6	Thm. 6.1 [A1, A2]
coherent_type_replacement	Protocol/Coherence/ SubtypeReplacementCore.lean	§6	Thm. 6.1 [A1, A2]
progress_conditions_ type_replacement	Protocol/Coherence/ SubtypeReplacementLiveness.lean	§6	Thm. 6.1 [A1, A2]
fifo_delivery_laws	Protocol/DeliveryModel.lean	§2	—
causal_delivery_laws	Protocol/DeliveryModel.lean	§2	—
lossy_delivery_laws	Protocol/DeliveryModel.lean	§2	—
handler_obligation_local	Runtime/Proofs/VM/BridgeStrengthening.lean	§7	Prop. 7.1 [A1]
handler_fragment_typing_ of_premises	Runtime/Proofs/VM/BridgeStrengthening.lean	§7	Prop. 7.1 [A1]
handler_vm_step_typing	Runtime/Proofs/VM/BridgeStrengthening.lean	§7	Prop. 7.1 [A1]
vm_bridge_soundness_composed	Runtime/Proofs/VM/BridgeStrengthening.lean	§7	Prop. 7.1 [A1]
effect_bisim_implies_ observational_equivalence	Runtime/Proofs/EffectBisim/Bridge.lean	§7	Prop. 7.1 [A1]
config_equiv_iff_ effect_bisim_silent	Runtime/Proofs/EffectBisim/ ConfigEquivBridge.lean	§7	Prop. 7.1 [A1]
byz_iface_wf_iff_coherent	Runtime/Proofs/Adapters/Distributed/ EnvelopeTheorems.lean	§7	Thm. 7.2 [A3]
bz1_byzantine_interface_ well_formedness	Runtime/Proofs/Adapters/Distributed/ EnvelopeTheorems.lean	§7	Thm. 7.2 [A3]
vm_byzantine_envelope_ adherence_of_witness	Runtime/Proofs/Adapters/Distributed/ EnvelopeTheorems.lean	§7	Prop. 7.4 [A3]
byzantine_cross_target_ conformance_of_witnesses	Runtime/Proofs/Adapters/Distributed/ EnvelopeTheorems.lean	§7	Prop. 7.4 [A3]
Runtime Gates			
canOperateUnderByzantineEnvelope	Runtime/Proofs/TheoremPack/API.lean	§7	Prop. 7.4 [A3]

Table E.1: Anchor index.