ICE-UTxO: Interleaving Coroutine Effects for Proof-Carrying UTxO Transactions

Charles Hoskinson
Input Output Group
Charles.Hoskinson@gmail.com

Abstract

Validators in the extended UTxO (eUTxO) model execute once per transaction: they cannot pause, resume, or interact with other on-chain components, leaving multi-step, multi-party coordination without direct ledger support.

We present ICE-UTxO, a conservative extension of eUTxO in which each transaction is a proof-carrying implementation of a multiparty session protocol. The model adds three layers—coroutine state on UTxOs, transaction-level sequencing via compiled Programmable Transaction Block programs, and Incrementally Verifiable Computation proof artifacts that gate commit. All three layers collapse to standard eUTxO when unused. Coordination scripts specify permitted interactions as MPST global types with event-structure semantics; PTB compilation produces deterministic schedules whose conformance is certified by IVC witnesses.

We mechanize the ledger-level operational semantics, conflict serializability, and MPST projection theorems in Lean 4. The development is fully constructive, uses no admitted lemmas or custom axioms, and avoids `Classical.choice`. Principal results include strong conflict serializability via a constructive bubble-sort proof, concurrent-to-serial refinement via stuttering simulation, and bidirectional MPST-to-ledger bridge theorems establishing that shard-local verification of projected traces is necessary and sufficient for global protocol consistency. Cryptographic primitives and the S-BAC consensus layer are modeled as oracles with explicit security assumptions.

To our knowledge, the serializability result is the first machine-checked proof of this property for a UTxO ledger with concurrent interleaving semantics.

Keywords: UTxO, multiparty session types, event structures, formal verification, Lean 4, zero-knowledge proofs

1 Introduction

Core idea. A transaction is a proof-carrying implementation of a multiparty protocol; the PTB program is the concrete schedule, and the IVC witness certifies its validity.

1.1 The Problem: Transactions as Single-Shot Validators

In the eUTxO model [Chakravarty et al.(2020)], a transaction consumes inputs, executes validators, and produces outputs. Each validator is a pure function: it receives a datum, a redeemer, and a read-only transaction context, and returns accept or reject. The model suits simple transfers but fails when applications need multi-step coordination across multiple UTxOs within a single atomic transaction.

Developers have converged on three workarounds, none satisfactory. *Transaction chaining* splits workflows across multiple transactions. This loses atomicity: intermediate states become

Table 1: Acronyms and abbreviations.

Acronym	Expansion
eUTxO	Extended Unspent Transaction Output
ICE	Interleaving Coroutine Effects
IVC	Incrementally Verifiable Computation
MPST	Multiparty Session Types
PCD	Proof-Carrying Data
PTB	Programmable Transaction Block
S-BAC	Sharded Byzantine Atomic Commit
ZK	Zero-Knowledge (proof)

visible on-chain, and any step can fail independently, enabling front-running and race conditions. *Monolithic validators* take the opposite approach, encoding the entire interaction in a single script. This preserves atomicity but loses modularity: the validator becomes a single point of complexity that cannot be composed from independent components. *Off-chain protocols* such as Hydra heads move interactions off-chain entirely, introducing additional trust assumptions and infrastructure requirements. None resolves atomicity, modularity, and verifiability simultaneously.

Multi-step interactions. Consider a DeFi protocol in which two UTxOs must interact across several resume/yield cycles within a single atomic transaction. For instance, a collateralized loan liquidation may require: (1) reading an oracle UTxO for a price feed, (2) computing a liquidation amount in the borrower's UTxO, (3) transferring collateral to a liquidator UTxO, and (4) updating the protocol's global state UTxO—all atomically. The eUTxO validator model has no notion of "pause and resume"; each validator executes exactly once, sees only its own datum, and cannot carry forward intermediate state.

In current eUTxO practice, a developer would encode this as a monolithic validator that inspects the full transaction context—functional but complex and non-reusable. Splitting into multiple transactions introduces race conditions (the oracle price may change between steps). Ethereum handles this via synchronous contract calls, but its account model sacrifices the parallel validation that UTxOs provide. ICE-UTxO offers an alternative: each component (oracle, borrower, liquidator, protocol state) remains an independent UTxO with its own validator, and the coroutine and effect system coordinates them within a single atomic transaction according to a session-typed protocol.

Cross-contract communication. A related limitation arises when two contracts need to exchange data during execution, not just at the transaction boundary. The eUTxO model forces developers into one of two unsatisfying choices: encode the entire multi-contract interaction as a single monolithic validator (sacrificing modularity), or split the interaction across multiple transactions (sacrificing atomicity). Neither preserves both properties.

Effect handling. A UTxO frequently needs to request a service (an oracle lookup, a token burn authorization, a permission check) and receive a result before continuing its computation. The validator model provides no mechanism for structured effects: there is no way for a validator to "raise" a request, have it handled by some external service within the same transaction, and then "resume" with the result.

Proof of coordination. Even if one could engineer multi-step interactions through clever datum encoding, there is no standard way to *prove* that the interleaving was valid: that the

schedule of operations respected causal dependencies and did not violate protocol invariants. The validity of the coordination is implicit in the validator logic, not an explicit, verifiable artifact.

To ground these limitations concretely, consider a cross-chain DEX aggregator. The system must atomically execute four steps: query an oracle for the current price of asset A in terms of asset B, check the user's collateral position against the quoted price, execute a swap on one liquidity pool if the price is favorable, and route to an alternative pool if not. All four must succeed or fail together—partial execution would expose the user's collateral to arbitrage.

In the current eUTxO model, encoding this requires either a monolithic validator that internalizes all routing logic (creating a maintenance burden and limiting composability with third-party pools), or a transaction chain that splits the steps across separate transactions (introducing race conditions where the price may change between steps). State channel approaches like Hydra could handle the interaction off-chain, but only among parties already sharing a Hydra head.

ICE-UTxO offers a third path: each component—oracle, collateral checker, liquidity pools—remains an independent UTxO with its own validator, coordinated by a session-typed protocol whose IVC proof certifies correct execution.

Today's eUTxO validators resemble sequential functions—powerful for their intended scope, but constrained when applications require richer interaction patterns. What is needed is a model where transactions are *concurrent programs with communication*, and where the schedule itself is a *verified artifact*.

1.2 The Insight: Transactions as Multiparty Protocols

The central idea is a change of perspective: ICE-UTxO treats each transaction as an instance of a *multiparty session protocol*. In standard eUTxO, a transaction is a batch operation—consume inputs, produce outputs, check validators. ICE-UTxO reframes this as a *conversation*: validators are participants in a protocol, and the transaction is one round of that protocol. A validator can pause (yield control and store its state), request a service (raise an effect), and later resume with the result. The rules governing who acts when are specified as a coordination script. This reframing resolves the limitations above by introducing three interlocking concepts:

UTxO coroutines are participants that can yield (pause execution) and resume, carrying their execution state as part of their UTxO datum. Concretely, each coroutine-enabled UTxO stores a *frame*—a tuple of program counter, local variables, method identifier, and a cryptographic hash binding the frame to its execution history (formalized in theorem 3.1). The UTxO lifecycle extends from `Created` through `Suspended_at_Yield` or `Suspended_at_Effect` to `Consumed` (the full lifecycle state machine is given in section 4.1, fig. 7). A coroutine that yields produces a new UTxO with an updated frame; a coroutine that is consumed has completed its participation in the transaction.

Effect handlers are dynamically-scoped services installed by the coordination script. When a coroutine raises an effect (e.g., "I need the current ETH/USD price"), the effect propagates to the nearest installed handler for that interface. The handler processes the effect and resumes the coroutine with a result. This is the *algebraic effects* pattern from programming language theory [Plotkin and Pretnar(2009)], adapted to the blockchain setting. Handlers are installed and uninstalled explicitly via transaction commands, and their lifetimes are bounded by the transaction scope.

Coordination scripts are *global types* in the sense of multiparty session types (MPST) [Honda et al.(2016)]. They specify the allowed interactions among roles as an *event structure*—a partial order of events

augmented with a conflict relation encoding mutual exclusion. The coordination script specifies the protocol; the compiled PTB program linearizes it into a deterministic schedule; the IVC witness certifies conformance.

Each component addresses a distinct concern. MPST global types specify the allowed multiparty coordination patterns, with well-studied safety and progress guarantees. To capture the independence of non-conflicting operations, event structures replace sequential traces with partial-order semantics. These specifications must then become executable: PTB-style compilation produces a concrete, deterministic format amenable to on-chain validation. Finally, IVC/PCD witnesses transform transactions into *proof-carrying artifacts*—each transaction carries a cryptographic certificate of its own validity, verified by an external ZK verifier whose soundness is an explicit assumption of the model (section 3.1).

Several of these constituent ideas have antecedents in both the blockchain and programming languages literatures. Multi-step contract execution on UTXO-like models has been explored by Ergo's multi-stage contracts [Chepurnoy and Saxena(2019)], which use transaction trees to encode sequential contract logic across multiple UTXO lifetimes, and by Sui's Programmable Transaction Blocks [Blackshear et al.(2023)], which compose multiple object operations within a single atomic transaction. The algebraic effects pattern is well-established in programming language theory [Plotkin and Pretnar(2009)] and has been applied to various computational settings, though not previously to on-chain UTXO validators. Session types for smart contracts were first explored by the Nomos system [Das et al.(2021)], which uses binary (two-party) session types based on linear logic to ensure resource safety in a concurrent contract language; ICE-UTxO extends this direction to multiparty protocols with arbitrarily many roles. Proof-carrying transactions in the blockchain setting were pioneered by ZEXE [Bowe et al.(2020)] and deployed in Aleo [Aleo(2021)], where each transaction carries a zero-knowledge proof of correct state transition; ICE-UTxO adapts this pattern to certify multiparty protocol compliance rather than arbitrary computation. The IVC proof mechanism builds on recent advances in folding schemes, particularly Nova [Kothapalli et al.(2022)], which makes incremental proof generation practical. The novelty of ICE-UTxO lies not in any single component but in their systematic integration: the demonstration that coroutines, algebraic effects, multiparty session types, PTB compilation, IVC proofs, and S-BAC compose coherently as a conservative extension of eUTxO, with the entire combination formally verified in Lean 4.

1.3 The Architecture: Three Layers on eUTxO

ICE-UTxO separates concerns into three layers, each addressing an independent verification question. Layer 1 asks: *can this UTxO resume its computation?* Layer 2 asks: *do these operations follow the declared protocol?* Layer 3 asks: *can we trust the execution without replaying it?* This separation allows each layer to be verified independently and composed via narrow interfaces.

Layer 1: Coroutine state on UTxOs. Each UTxO optionally carries a *frame* $(pc, locals, methodId, hash)$. The frame records the coroutine's suspension point, enabling it to be resumed in a future transaction step. This is the *process layer*—it provides the mechanism for multi-step execution.

Layer 2: Transaction-level coordination. Mechanism alone is not enough; we also need a specification of what coordination is allowed. A coordination script (global type) is compiled to a PTB-style program—a sequence of commands with dataflow through temporary result registers $Result(i)$. This is the *protocol layer*—it constrains which interleavings are valid.

Layer 3: IVC proof artifacts. Given mechanism (Layer 1) and specification (Layer 2), the remaining question is trust: how does the ledger verify that the mechanism followed the

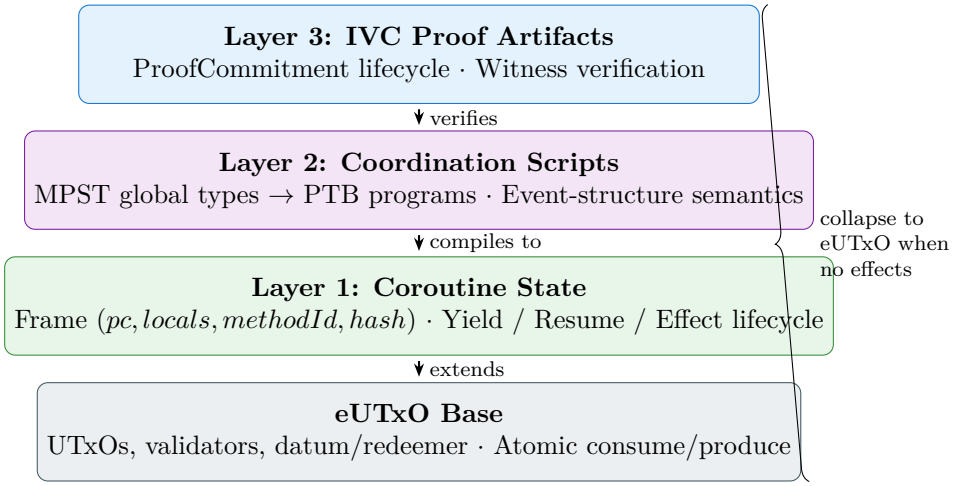

Figure 1: Architecture layers of ICE-UTxO. Layer 1 (coroutine state) extends UTxOs with frames; Layer 2 (coordination) compiles MPST global types to PTB programs; Layer 3 (verification) attaches IVC proof artifacts. When no coroutines yield and no effects are raised, all three layers collapse and ICE-UTxO degenerates to standard eUTxO.

Table 2: Mapping from Chainspace to ICE-UTxO.

Chainspace	**ICE-UTxO**
Object	Frame-carrying UTxO
Procedure bundle	PTB program + IVC witness
Checker	Witness verifier + ledger checks
S-BAC	Cross-shard atomic commit

specification? The transaction carries proof commitments that certify the interleaving trace conforms to the coordination script. Validators check the proof rather than re-execute the coroutine logic. This is the *verification layer*.

Conservative extension. When all three layers collapse—no coroutines yield, no effects are raised, no coordination script is needed—ICE-UTxO degenerates to standard eUTxO. Concretely, every standard eUTxO transaction is a valid ICE-UTxO transaction with empty frame fields, no effect handlers, and a trivial coordination witness; the new machinery adds capability without breaking compatibility. The formal statement appears in section 4.5.

1.4 Deployment Model: S-BAC for Cross-Shard Atomicity

When a transaction touches only *owned objects* (coroutine frames belonging to a single party), it can proceed on a fast path without full consensus. When it touches *shared objects* (effect handlers, shared state across shards), it requires consensus for atomic commit. ICE-UTxO adopts the Sharded Byzantine Atomic Commit (S-BAC) protocol from Chainspace [Al-Bassam et al.(2018)]:

- Each shard checks its *local projection* of the coordination script during the prepare phase.
- The IVC witness lets shards validate the interleaving without re-executing private computation.
- If all shards prepare successfully, the transaction commits; if any shard aborts, all abort.

1.5 Formal Verification: Zero Sorry, Zero Axioms

The ICE-UTxO ledger-level semantics have been formalized in Lean 4 in about 4,500 lines of code across eleven source files (including `test_axioms.lean`) plus `lakefile.lean`, organized in four core modules (`StarstreamPilot.lean`, `Script.lean`, `PTB.lean`, `SBAC.lean`) and supporting oracle modules. Cryptographic primitives and the S-BAC consensus layer are modeled as oracles with explicit assumptions (section 3.1). The mechanization achieves:

- **Zero sorry**: no admitted lemmas anywhere in the development.
- **Zero custom axioms**: only the standard Lean kernel axioms (`propext`, `Quot.sound`, `funext`).
- **Minimal classical reasoning**: `Classical.choice` is never invoked, and the development uses a single localized `classical` case split; the proofs are otherwise constructive.

1.6 Contributions

This paper makes the following contributions:

1. **ICE-UTxO model.** A conservative extension of eUTxO with coroutines, algebraic effects, and proof-carrying transactions (sections 3 and 4).
2. **Coordination scripts.** A formal language for multiparty coordination based on MPST global types with event-structure semantics (section 3).
3. **PTB compilation.** Translation from coordination scripts to PTB-style bytecode—adapting Sui's Programmable Transaction Block format [Blackshear et al.(2023)]—with explicit dataflow and formal correctness guarantees (section 3).
4. **S-BAC integration.** Shard-local verification using projected coordination scripts, composing Chainspace's S-BAC protocol [Al-Bassam et al.(2018)] with session-type witnesses to enable protocol-aware cross-shard atomic commit (sections 3 and 6).
5. **Lean 4 mechanization.** Complete formal verification with zero sorry, zero custom axioms, and mostly constructive proofs (section 7).
6. **Strong serializability proof.** A constructive bubble-sort proof that acyclic full-conflict precedence graphs imply all conflict-respecting permutations produce the same core state (section 5).
7. **Systematic integration.** A demonstration that coroutines, algebraic effects, MPST, PTB compilation, IVC proofs, and S-BAC compose coherently as a conservative extension of eUTxO, with the combination verified end-to-end. Individual components draw on established techniques (table 6 in section 8); the contribution is their integration within a single formally verified framework.

The contributions are formal: we establish the model and prove its safety properties via mechanized proof. Empirical evaluation of proving costs, transaction throughput, and validation latency under realistic workloads is orthogonal and left to implementation work.

1.7 How to Read This Paper

This paper serves multiple audiences with different interests. The following guide maps reader backgrounds to relevant sections.

For blockchain developers: sections 1, 3.1, 3.2, 3.4 and 4.1 to 4.3 introduce the model and its operational behavior.

For formal methods researchers: sections 3.3, 5 and 6 contain the main theoretical contributions. Section 7 documents the Lean mechanization.

For protocol designers: sections 3.6, 4.2 and 6.2 describe the deployment architecture and its trust boundaries.

1.8 Paper Roadmap

Section 2 reviews background. Section 3 presents the ICE-UTxO model in full. Section 4 defines the operational semantics and proves ledger safety invariants. Section 5 establishes strong conflict serializability. Section 6 bridges the MPST coordination layer to the ledger commit mechanism. Section 7 discusses the Lean 4 mechanization. Section 8 surveys related work. Section 9 discusses limitations and future directions. Section 10 concludes.

2 Background

ICE-UTxO combines five ideas from different research communities: the eUTxO ledger model provides the state substrate; multiparty session types specify coordination protocols; event structures give those protocols a partial-order semantics; Programmable Transaction Blocks supply a deterministic execution format; and Sharded Byzantine Atomic Commit handles cross-shard consistency. Each addresses a distinct concern; none suffices alone. The following subsections review each foundation in turn.

2.1 Extended UTxO

Bitcoin's original UTxO model represents value as *unspent transaction outputs*: discrete "coins" that are created by one transaction and consumed by another. Spending a UTxO requires providing a witness (typically a digital signature) that satisfies the locking script attached to that output. While elegant and naturally parallelizable—distinct UTxOs can be validated independently—the original model is limited in expressiveness: there is no persistent state beyond the locked value itself, and the scripting language (Bitcoin Script) is intentionally non–Turing-complete. These constraints make it difficult to encode smart contracts that maintain state across interactions, perform rich computation, or coordinate multiple parties. The *extended UTxO* (eUTxO) model [Chakravarty et al.(2020)] was designed to bring smart contract capability to UTxO-based blockchains while preserving the parallelism and deterministic validation that make the UTxO approach attractive. By contrast, Ethereum's account model maintains global mutable state that contracts read and write during execution, enabling rich programmability but introducing sequential execution dependencies, reentrancy vulnerabilities, and non-deterministic gas consumption.

The eUTxO model augments each UTxO with a typed *datum* (persistent state) and each spending transaction with a *redeemer* (an argument provided by the party constructing the transaction). Each *validator* functions as a pure deterministic gate:

$$\mathit{validator} : \mathit{Datum} \times \mathit{Redeemer} \times \mathit{TxContext} \to \mathit{Bool}$$

where *TxContext* provides a read-only view of the transaction's inputs, outputs, validity interval, fee structure, minting policies, and other metadata. Validators are pure functions: given the same datum, redeemer, and context, they always return the same Boolean. This purity

enables predictable validation—transaction outcomes can be predicted off-chain before submission, validators can be executed in parallel across distinct UTxOs, and formal verification of validator logic is tractable. Cardano implements eUTxO with Plutus validators, where datums and redeemers are serialized as a universal data type and validators are compiled to Plutus Core (a variant of System F).

Despite these strengths, the original eUTxO model has limitations. The result: atomicity without modularity, or modularity without atomicity. The most significant limitation is the "consume to read" problem: inspecting a UTxO's datum requires including it as a transaction input, which *consumes* it. This forces contention—if two transactions need to read the same datum (e.g., an oracle price feed), they must compete for the UTxO, serializing access. A related issue is the *double satisfaction* problem [Chakravarty et al.(2020)]: when a transaction has multiple script inputs, a single output may inadvertently satisfy the validation conditions of two different validators, allowing one validator's intended output to be "stolen" by another. Furthermore, multi-step interactions between parties require multiple on-chain transactions across multiple blocks, breaking atomicity and exposing intermediate states to front-running or abandonment.

Several Cardano Improvement Proposals have partially addressed these limitations. CIP-31 [Cardano Improvement Proposal(2022)] introduces *reference inputs*, allowing a transaction to read a UTxO's datum without consuming it—solving the consume-to-read problem for oracle feeds and shared configuration data. CIP-32 introduces *inline datums*, storing data directly in the UTxO output rather than as a hash, reducing the need for off-chain datum management. CIP-33 introduces *reference scripts*, enabling transactions to reference validator scripts stored on-chain rather than re-attaching them to every transaction, significantly reducing transaction size. Together (deployed in the Vasil hard fork), these extensions improve usability and reduce contention, but they still cannot express multi-step atomic protocols—sequences of dependent operations that must all succeed or all fail—within a single transaction.

Other UTxO-based platforms have explored related extensions. Ergo [Ergo Platform(2019)] supports *data inputs* (analogous to CIP-31 reference inputs) from its inception and enables multi-stage contracts via ErgoScript, a language based on Sigma protocols that supports non-interactive zero-knowledge proofs natively. Bartoletti and Zunino's BitML [Bartoletti and Zunino(2018)] provides a process calculus for Bitcoin smart contracts with formal compilation to standard Bitcoin transactions, demonstrating that surprisingly rich contract logic can be encoded even in Bitcoin's restricted scripting model. More recently, proposals for Bitcoin *covenants*—notably BIP-119 (OP_CHECKTEMPLATEVERIFY)—would allow a UTxO's spending conditions to constrain the *outputs* of the spending transaction, propagating conditions forward in the UTxO graph. Vinogradova and Melkonian [Vinogradova and Melkonian(2024)] explore message-passing semantics for eUTxO, providing a communication-oriented perspective on inter-UTxO interaction. ICE-UTxO builds on the post-Vasil eUTxO model and extends it further with coroutines and algebraic effects, enabling multi-step atomic protocols within a single transaction while retaining the deterministic, parallelizable character of UTxO validation.

2.2 Multiparty Session Types

The eUTxO extensions above enable richer computation within individual UTxOs, but they do not provide a formal coordination language for multi-party interactions. Session types fill this gap. They specify communication protocols among multiple parties, with mechanical safety guarantees: a well-typed protocol cannot deadlock, and no participant can send the wrong message at the wrong time. They function as typed contracts for message exchange, analogous to function type signatures that prevent calling a function with incorrect argument types. Originally developed for binary (two-party) sessions by Honda et al., the theory was extended to the multiparty setting (MPST) by Honda, Yoshida, and Carbone [Honda et al.(2016)], enabling protocols involving three or more roles with complex interaction patterns including del-

egation, branching, and recursion. The MPST framework has since been applied to verify communication-intensive systems including web services, operating system kernels, and distributed algorithms [Hüttel et al.(2016)].

A *global type* G specifies the complete protocol from an omniscient perspective. For instance, if role p sends a price request to role q, who then responds with a price, the global type records both steps in sequence. The formal syntax captures this as a grammar:

$$G ::= p \to q : \langle S \rangle.G \mid G_1 + G_2 \mid \mu X.G \mid X \mid \mathbf{end}$$

The constructor $p \to q : \langle S \rangle.G$ means that role p sends a message of payload type S to role q, after which the protocol continues as G. The choice $G_1 + G_2$ represents a branching point where the protocol may proceed along either branch (typically determined by a designated role). The recursion $\mu X.G$ binds the variable X in G, enabling repeating protocols such as auction rounds or heartbeat exchanges; **end** terminates the session. *Projection* extracts a *local type* $L_r = G \upharpoonright r$ for each role r, describing only that role's own sends and receives. If every participant faithfully follows its projected local type, the global protocol specification is satisfied.

The MPST metatheory establishes three key safety properties [Bettini et al.(2008), Coppo et al.(2013), Scalas and Yoshida(2016)]. *Communication safety* ensures that no type mismatch can occur: when a role sends a message of type S, the receiving role expects exactly type S. *Protocol conformance* guarantees that each party's runtime behaviour is an instance of its projected local type—no party can deviate from the prescribed interaction sequence. *Progress* (deadlock-freedom) ensures that well-typed sessions always advance: no configuration arises where every role is waiting for a message that will never be sent. Progress typically requires well-formedness conditions on the global type (e.g., no mixed choices where different roles independently decide different branches) and fairness assumptions (messages are eventually delivered).

In ICE-UTxO, session types are applied in a setting that differs fundamentally from the traditional distributed-processes-over-channels model. Sessions are scoped to a single atomic transaction: all protocol steps occur within one transaction, not across multiple blocks or network round-trips. This has a simplifying consequence for progress—because a transaction either executes completely or not at all (atomicity), "message delivery" is guaranteed within the session scope, and progress reduces to an all-or-nothing property rather than requiring ongoing fairness. Each UTxO participating in the protocol plays a role in the session; its validator enforces the projected local type by checking that the transaction's structure matches the expected communication pattern. Adversarial deviation—a participant attempting to execute a transaction that violates the protocol—is prevented by proof-carrying semantics (section 3): each transaction carries cryptographic evidence that the executed schedule conforms to the coordination script.

The MPST framework has been extended in numerous directions: asynchronous communication with bounded buffers [Scalas and Yoshida(2016)], timed sessions with deadline constraints, session delegation (passing a session endpoint to another party), and session subtyping. Of particular relevance to ICE-UTxO is the event-structure semantics for MPST developed by Castellani et al. [Castellani et al.(2023)], which interprets global types not as sequential traces but as event structures (section 2.3). This interpretation captures the *concurrency* inherent in multiparty protocols: when two communication actions involve disjoint sets of roles, they are genuinely independent events rather than arbitrarily interleaved sequential steps. ICE-UTxO builds directly on this event-structure semantics to derive correct-by-construction schedules for multi-UTxO coordination.

2.3 Event Structures

MPST global types specify *what* interactions are allowed, but not *how concurrent* those interactions are—whether two protocol steps must happen sequentially or can occur in any order. Event structures answer this question. Blockchain transactions involving multiple UTxOs often

contain operations that are *causally dependent*—one must complete before another can begin (e.g., reading an oracle price before computing a liquidation amount)—as well as operations that are genuinely *independent* (e.g., two unrelated token transfers within the same transaction). A simple sequential model does not capture this distinction: it imposes a total order where none is required, obscuring which operations can be safely reordered. Event structures, introduced by Winskel [Winskel(1987)], provide a mathematical framework that models both causal order and mutual exclusion (*conflict*) explicitly, enabling precise reasoning about concurrency. This is critical for ICE-UTxO's serializability results: by tracking which operations are independent, we can prove that different valid execution orders produce the same outcome.

Formally, an *event structure* is a triple $(E, \leq, \#)$ where E is a set of events, $\leq \subseteq E \times E$ is a partial order representing *causality* (if $e_1 \leq e_2$, then e_1 must occur before e_2), and $\# \subseteq E \times E$ is a symmetric, irreflexive *conflict* relation, subject to *hereditary conflict*: if $e_1 \# e_2$ and $e_2 \leq e_3$, then $e_1 \# e_3$ (conflict propagates forward along causality). A *configuration* $C \subseteq E$ is a set of events that is conflict-free ($\neg(e_1 \# e_2)$ for all $e_1, e_2 \in C$) and down-closed (if $e \in C$ and $e' \leq e$, then $e' \in C$). Intuitively, a configuration represents a consistent snapshot of "what has happened so far"—every event's prerequisites are present, and no two mutually exclusive events have both occurred. An event e is *enabled* in configuration C if $e \notin C$, all predecessors of e under $\leq$ are in C, and e does not conflict with any event in C. A *valid trace* is a sequence of events $e_1, e_2, \ldots, e_n$ such that each prefix $\{e_1, \ldots, e_k\}$ forms a configuration—equivalently, a chain of configurations from $\emptyset$ to the final state.

Castellani et al. [Castellani et al.(2023)] showed that MPST global types can be interpreted as event structures: each communication action ($p \to q : \langle S \rangle$) becomes an event; causal dependencies arise from the protocol's sequential structure (a send must precede its continuation); and conflict arises from choice (the two branches of $G_1 + G_2$ are in conflict, since only one branch can be taken). This gives MPST a *concurrency semantics* rather than an interleaving semantics. The distinction matters for ICE-UTxO because multiple UTxO operations within one transaction may be genuinely independent—their relative order is irrelevant to the outcome—and the event-structure representation captures this independence explicitly, enabling tighter correctness proofs.

Any acyclic event structure can be *linearized*: its events can be topologically sorted into a total order that respects causality. When two events are *concurrent* (neither causes the other, and they are not in conflict), they may appear in either order in a linearization—and, crucially, the resulting state is the same regardless of order. This *confluence* property is the foundation of the serializability results in section 5. The PTB compilation (section 3.5) performs exactly this linearization, producing a deterministic command sequence from the event structure's partial order, with explicit dataflow through result registers ensuring that causal dependencies are respected.

Figure 2 illustrates a small event structure arising from a coordination script. Event e_1 (install a handler) causally precedes both e_2 (resume coroutine U_1) and e_3 (resume coroutine U_2). Event e_2 causally precedes e_4 (raise an effect), which in turn precedes e_5 (handle the effect). The dashed line between e_3 and e_4 denotes conflict: they cannot both appear in the same configuration. This models a protocol where U_1 and U_2 cannot both be active simultaneously—the coordinator must choose which coroutine to resume, and the unchosen path is excluded from the execution.

2.4 Programmable Transaction Blocks

In standard eUTxO (and Bitcoin), a transaction is a single atomic step: consume inputs, produce outputs. There is no way to express "first read this UTxO, then use the result to compute something, then write to another UTxO" as sequential steps within one transaction. Each validator runs independently, seeing only its own datum, redeemer, and the transaction context—there is no mechanism for one validator's output to flow as input to another validator within the same

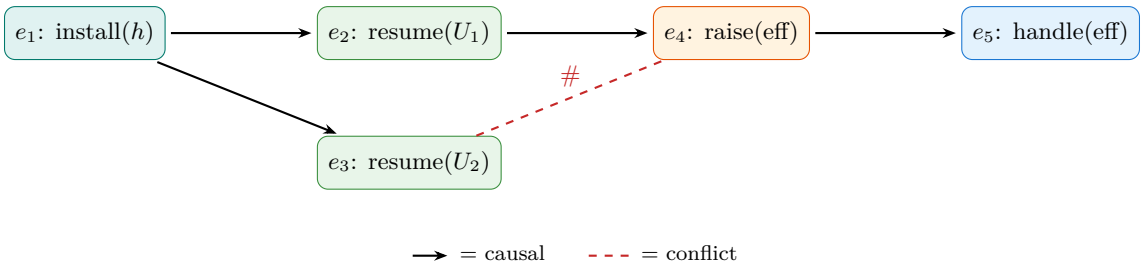

Figure 2: Example event-structure DAG. Solid arrows denote causal order ($<$); the dashed line between e_3 and e_4 denotes conflict (#).

transaction. This limitation prevents expressing multi-step atomic protocols where intermediate results must be threaded through a sequence of operations. *Programmable Transaction Blocks* (PTBs), introduced by Sui [Blackshear et al.(2023)], address this gap by allowing a transaction to contain an ordered sequence of commands with explicit dataflow between them. Move's transaction scripts [Blackshear et al.(2019)] are a predecessor, allowing multiple operations within a single transaction but without the register-based dataflow model.

A PTB is a list of commands $[c_0, c_1, \ldots, c_n]$. Each command c_i performs an operation—calling a Move function, reading an on-chain object, splitting or merging coins, transferring ownership—and stores its result in a register $\mathit{Result}(i)$. Subsequent commands may reference $\mathit{Result}(j)$ for any $j < i$ as input, creating explicit dataflow edges from producers to consumers. All commands execute sequentially within one atomic transaction: if any command fails (e.g., an assertion violation or insufficient balance), the entire transaction is reverted with no observable effect. There are no loops or conditional branches within a PTB; any conditional logic must be resolved at transaction construction time by choosing which commands to include. This design keeps the execution model simple and predictable while enabling rich composition of on-chain operations.

Other blockchain platforms achieve multi-step execution through different mechanisms. Ethereum handles this implicitly via smart contract code: a single external call can trigger a chain of internal calls across contracts, but the "steps" are EVM opcodes, not user-visible commands, and composing across contracts requires knowledge of their call interfaces and careful reasoning about reentrancy. Solana transactions contain multiple *instructions* targeting different programs, executing sequentially with shared account state—conceptually similar to PTBs but without the explicit result-register dataflow model. Aptos supports *Move scripts* that call multiple entry functions in sequence within a single transaction. ICE-UTxO adopts the PTB model as a compilation target because it provides an explicit, inspectable, deterministic schedule—properties that are well-suited for formal verification and proof generation.

In ICE-UTxO, coordination scripts (derived from MPST global types) are compiled to PTB-style programs. The event structure's partial order is linearized into a command sequence with explicit dataflow through result registers: if event e_j causally depends on event e_i, then command c_j references $\mathit{Result}(i)$. This compilation step is formalized in section 3.5, with correctness theorems ensuring that the PTB faithfully implements the coordination script's semantics—every valid execution of the PTB corresponds to a valid trace in the event structure, and vice versa.

2.5 Sharded Byzantine Atomic Commit (S-BAC)

When a blockchain's state is *sharded*—partitioned across independent validator sets, each responsible for a disjoint subset of UTxOs—a transaction that touches UTxOs on multiple shards cannot be validated by any single shard alone. Without a coordination protocol, partial execution becomes possible: one shard commits its portion of the transaction while another rejects,

leaving the ledger in an inconsistent state where some inputs are consumed but the corresponding outputs are never created. Atomic commit protocols solve this problem by ensuring *all-or-nothing* semantics: either every shard applies the transaction, or no shard does.

Chainspace [Al-Bassam et al.(2018)] introduced *Sharded Byzantine Atomic Commit* (S-BAC), adapting classical two-phase commit for the Byzantine fault-tolerant setting. The protocol operates in two phases. In the *prepare* phase, the transaction is sent to all relevant shards; each shard runs BFT consensus (e.g., PBFT with $f < n/3$ Byzantine tolerance) among its validators to check local validity—inputs exist and are unspent, all validator scripts pass—and tentatively locks the affected inputs, preventing conflicting transactions from proceeding. Each shard votes *prepare* (local validity confirmed) or *abort* (local check failed). In the *accept* phase, if all shards vote prepare, the coordinator broadcasts *commit* and every shard applies the transaction; if any shard votes abort, the coordinator broadcasts *abort* and every shard discards the tentative state and releases locks.

S-BAC guarantees *safety unconditionally*: no partial commits can occur, even under Byzantine behaviour by up to f validators per shard. If a shard's honest majority votes prepare, the locked inputs will not be double-spent; if any shard votes abort, no shard will commit. *Liveness* requires partial synchrony and honest supermajorities: a fully malicious shard can force aborts (but not partial commits), and network delays can stall the protocol until timeouts trigger abort. In practice, aborted transactions can be retried. The abort rate under adversarial conditions is a performance concern—it affects throughput and latency—but not a safety concern, since the invariant that committed transactions are fully valid is never violated.

Several alternative approaches to cross-shard or cross-chain atomicity exist. OmniLedger's Atomix protocol [Kokoris-Kogias et al.(2018)] uses a similar two-phase structure with client-driven coordination and lock-unlock semantics. Cosmos IBC provides asynchronous cross-chain messaging with timeout-based rollback, trading strong consistency for availability. Polkadot's relay-chain model achieves cross-shard ("parachain") finality through unified consensus at the relay layer. ICE-UTxO adopts S-BAC for its strong consistency guarantees: the all-or-nothing commit semantics ensure that coordination scripts—which may span multiple shards—either execute completely or not at all, aligning with the atomicity required by MPST session semantics.

In ICE-UTxO, S-BAC is augmented with proof-carrying semantics. During the prepare phase, each shard validates its local projection of the coordination script and checks the IVC witness attached to the transaction. The witness provides cryptographic assurance that the cross-shard data is consistent—that the commands referencing UTxOs on other shards were executed correctly according to the coordination script—without requiring shards to re-execute computation performed on remote state. This means shards do not need to trust each other beyond the BFT assumption; the proof artifacts substitute for trust. The formal integration of S-BAC with the coordination model and proof layer is detailed in section 3.6.

Summary. These five foundations address distinct concerns in ICE-UTxO's architecture: eUTxO provides the state model; MPST global types with event-structure semantics provide the protocol specification language; PTB compilation provides a deterministic execution format; and S-BAC provides cross-shard atomic commit. The remaining ingredient—not an external foundation but ICE-UTxO's central contribution—is *proof-carrying semantics*: each transaction carries IVC proof artifacts that certify the executed schedule conforms to the coordination script. This verification layer, introduced in section 3, is what enables on-chain enforcement of protocol compliance without requiring trust between participants or re-execution of private computation.

3 The ICE-UTxO Model

This section presents the formal ICE-UTxO model, built in five steps. First: state components—what the ledger holds (section 3.1). Second: coordination scripts—what interactions are allowed (section 3.2). Third: event-structure semantics—what correctness means (section 3.3). Fourth: PTB compilation—how scripts become programs (section 3.5). Fifth: S-BAC integration—how shards synchronize (section 3.6). Each step answers one question; together they give the complete model.

Readers familiar with eUTxO may focus on section 3.1 (Definitions 3.1–3.7) to see what is new in the state model, then skip to section 3.5 for the compilation pipeline. Readers from the session-types community may focus on sections 3.2 to 3.4 for the event-structure semantics and projection theory.

Standard eUTxO validators execute once and terminate—they cannot pause, request external services, or prove that their coordination with other validators was correct. The state components defined below extend the eUTxO ledger with the machinery needed to support these three capabilities: coroutine suspension (frames), structured service requests (effects and handlers), and cryptographic verification of protocol conformance (proof commitments).

3.1 State Components

Where the standard eUTxO model has only UTxOs and transactions, ICE-UTxO adds five new concepts, introduced in dependency order: **frames** (coroutine suspension state), **effects** and **handlers** (structured service requests and their processors), **proof commitments** (IVC certificates tracking verification status), and augmented **transactions** and **ledger** with effect queues and handler stacks.

We use the following identifier domains, all drawn from $\mathbb{N}$: *UTXOId*, *TxId*, *InterfaceId*, *ProcessId*, *CommitmentHash*. The formalization does not impose size limits on frames or UTxO datums; in a deployment, maximum frame size would be bounded by transaction size limits, and storage costs would be proportional to frame size.

The first concept unique to ICE-UTxO is the *coroutine frame*. In conventional programming, when a function is called, the runtime saves a return address and local variables on a stack frame; when the function returns, the frame is popped and execution resumes at the saved address. A coroutine frame plays an analogous role for UTxO coroutines. When a coroutine yields (pauses execution mid-transaction), it serializes its program counter, local variable bindings, and method identifier into a frame, which is stored as part of the UTxO's on-chain datum. When the coroutine is later resumed, the frame is read back from the datum and execution continues from the saved suspension point.

This mechanism is necessary because standard eUTxO validators are single-shot: a validator runs once, returns accept or reject, and is done. ICE-UTxO needs validators that can pause mid-execution, produce an intermediate UTxO carrying the suspension state, and be resumed later—potentially multiple times—within the same atomic transaction. The frame is the data structure that makes this possible. Without it, there would be no way to carry forward the execution context between successive steps of a multi-step interaction.

Definition 3.1 (Frame)**.** *A frame is a tuple* $f = (pc, locals, methodId, hash)$ *where* $pc : \mathbb{N}$ *is the program counter (instruction index within the coroutine body),* $locals : List(Value)$ *stores the coroutine's local variable bindings,* $methodId : \mathbb{N}$ *identifies which method or entry point is being executed, and* $hash : CommitmentHash$ *is a cryptographic commitment chaining this frame to the preceding frame in the coroutine's execution history.*

The *hash* field deserves special attention. Each frame's hash is computed over the tuple $(previous\_hash, pc, locals, methodId)$, forming a hash chain analogous to a blockchain's block hash chain. This ensures that when a coroutine is resumed, the resuming logic can verify that

it is continuing from a genuine prior suspension rather than a forged state. An adversary who tampers with a suspended frame—changing the program counter to skip a check, for instance—would produce a hash that does not match the expected chain, and the verification would fail. In the formal model, frame integrity is guaranteed by the IVC proof verification layer (Layer 3, section 4.2) rather than checked by the ledger directly; the ledger trusts the proof that the frame chain is intact.

The next two definitions introduce *effects* and *handlers*, which together implement the algebraic effects pattern from programming language theory, adapted to the blockchain setting.

In traditional programming, a function might throw an exception or perform I/O—these are "effects" in the PL sense. Algebraic effects [Plotkin and Pretnar(2009)] generalize this idea: a computation can "raise" a structured request (e.g., "I need the current ETH/USD price") and be suspended until the request is fulfilled by a handler. In ICE-UTxO, effects model structured interactions between a UTxO coroutine and an external service within a single atomic transaction. The effect triple $(iface, source, tag)$ identifies what service is needed ($iface$), who is asking ($source$), and what specific operation is requested (tag).

Definition 3.2 (Effect)**.** *An effect is a triple $e = (iface, source, tag)$ where $iface$: InterfaceId identifies the target interface, $source$: UTXOId identifies the raising coroutine, and tag : $\mathbb{N}$ distinguishes effect operations.*

A handler is the counterpart to an effect: it is a service that has been "installed" for a particular interface within the current transaction scope. When a coroutine raises an effect targeting interface i, the system dispatches the request to the handler currently installed for i. Handlers are organized in a stack per interface: installing a handler pushes it onto the stack, uninstalling pops it. The topmost handler receives dispatched effects. This stack discipline mirrors the scoping rules of algebraic effect handlers in programming languages like Eff or Koka, where the innermost enclosing handler for a given effect type is the one that intercepts the raised effect.

Definition 3.3 (Handler)**.** *A handler is a pair $h = (iface, hid)$ where $iface$ identifies the interface and hid is a unique handler identifier.*

To make this concrete: in the liquidation scenario from section 1.1, the coordinator installs a `priceHandler` for the oracle interface. When the borrower's coroutine raises a `getPrice` effect, it is dispatched to `priceHandler`, which provides the current ETH/USD price and resumes the borrower's coroutine with the result. The entire exchange—install handler, raise effect, dispatch, resume with result—occurs within a single atomic transaction.

The final state component before transactions and the ledger is the *proof commitment*, which connects the execution layer to the verification layer.

In a sharded system, not every shard can re-execute every coroutine's computation. Instead, the transaction author generates a cryptographic proof certifying correct execution, attached as a "proof commitment" that progresses through a lifecycle: NotStarted → Generating → Verifying → Verified (with failure edges to Failed). Only when all commitments reach Verified can the transaction commit. The analogy to proof-carrying code is direct: just as PCC attaches a proof of safety to a mobile program so the host can verify without re-executing, ICE-UTxO attaches a proof of protocol conformance to each transaction so validators can verify without replaying the full coroutine logic. Remark 3.5 details what the proof certifies.

Definition 3.4 (Proof Commitment)**.** *A proof commitment records the status of an IVC/PCD (proof-carrying data) proof:*

$$p = (proofKind, processId, commitHash, verifyKey, phase, stepNumber)$$

where $proofKind \in \{IVC\_Step, IVC\_Accumulator, Witness\}$ and phase follows:

$$NotStarted \to Generating \to Verifying \to Verified$$

with failure edges from Generating and Verifying to Failed. The commitHash field binds the proof to the transaction context; it is intended to be computed as a hash over the transaction's inputs, outputs, and coordination witness, providing anti-replay protection. The exact hash input is not formalized in the current mechanization (see section 4.2, Security Assumption 1).

Remark 3.5 (IVC Proof Predicate)**.** *The transaction author generates an IVC proof; on-chain validators verify it without re-executing the underlying computation. If verification fails, the proof commitment transitions to* `Failed` *and the transaction cannot commit (section 4.2). Soundness of this mechanism rests on Security Assumption 1 (ZK verifier soundness, section 4.2): the model proves that* if *the verifier is honest,* then *only correctly executed transactions can commit.*

Concretely, the IVC proof for a transaction tx with coordination witness $W = (S, tr)$ certifies the conjunction of three predicates:

1. ***Execution correctness.*** *Each coroutine step in the execution trace produced the correct output frame from the input frame according to the coroutine's program logic.*
2. ***Effect resolution.*** *Every effect raised during execution was dispatched to the handler at the top of the appropriate interface stack, and the handler's return value was correctly propagated to the raising coroutine.*
3. ***Protocol conformance.*** *The coordination witness satisfies $witnessGlobalOK(W)$: the trace tr is a valid trace of the script S (theorem 3.39).*

The commitHash field binds this proof to a specific transaction context by hashing the transaction's inputs, outputs, read set, and coordination witness. A valid proof under verifyKey cannot be reused for a different transaction because the hash input would differ, invalidating the proof. The exact hash construction is a deployment parameter; the model abstracts it via Security Assumption 1 (section 4.2).

With frames, effects, handlers, and proof commitments in hand, we can now define the ICE-UTxO transaction and ledger. A standard eUTxO transaction has inputs and outputs. ICE-UTxO extends this with several new fields: *readSet*—UTxOs that are read but not consumed, analogous to Cardano's reference inputs and Hyperledger Fabric's read sets for MVCC validation; *writeSet*—UTxOs that will be modified; *proofCommitments*—the IVC proof artifacts described above; and *phase*—a lifecycle state tracking the transaction's progress through the commit protocol.

Definition 3.6 (Transaction)**.** *A transaction is:*

$$tx = (id, inputs, outputs, readSet, writeSet, proofCommitments, phase)$$

where $inputs, outputs, readSet, writeSet : \mathcal{P}_{fin}(UTXOId)$ (readSet enables pure observation of UTxOs without consuming them, analogous to Cardano's CIP-31 reference inputs and Hyperledger Fabric's read sets for multi-version concurrency control), $proofCommitments : List(ProofCommitment)$, and $phase \in \{Idle, Reserve, Executing, Committing, Committed, Rollback, Failed\}$.

The augmented ledger state extends the standard UTxO ledger from a passive record of which outputs exist into an active execution environment capable of managing concurrent multi-step transactions. Standard UTxO ledgers track which outputs exist (*utxos*) and which have been consumed. ICE-UTxO adds: *locked* (outputs reserved by in-flight transactions, enabling concurrency control), *pending* (transactions currently being processed), *effects* (per-interface queues of pending service requests), and *handlerStacks* (per-interface stacks of installed service handlers). These additions are what allow the ledger to support the coroutine, effect, and coordination mechanisms defined above.

Table 3: State components of the ICE-UTxO ledger.

Component	Type	Purpose
utxos	$\mathcal{P}_{\text{fin}}(UTXOId)$	Currently unspent outputs
consumed	$\mathcal{P}_{\text{fin}}(UTXOId)$	Previously spent outputs (monotonically growing)
locked	$\mathcal{P}_{\text{fin}}(UTXOId)$	Outputs reserved by in-flight transactions
pending	$\mathcal{P}_{\text{fin}}(Tx)$	Transactions in progress (not yet committed/aborted)
history	$\text{List}(Tx)$	Committed transactions in order
effects	$InterfaceId \to \text{List}(Effect)$	Pending service request queues
handlerStacks	$InterfaceId \to \text{List}(Handler)$	Installed handler stacks (LIFO per interface)

Definition 3.7 (Ledger)**.** *The ledger state is:*

$$L = (utxos, consumed, locked, pending, history, effects, handlerStacks)$$

where:

- $utxos, consumed, locked : \mathcal{P}_{fin}(UTXOId)$
- $pending : \mathcal{P}_{fin}(Tx)$
- $history : List(Tx)$ *(committed transactions in order)*
- $effects : InterfaceId \to List(Effect)$ *(pending effect queues)*
- $handlerStacks : InterfaceId \to List(Handler)$ *(installed handler stacks)*

The effects and handler stacks are organized per-interface as stacks, supporting dynamic installation and uninstallation of handlers. When a coroutine raises an effect on interface i, it is dispatched to the handler at the top of $handlerStacks(i)$; if the stack is empty, the effect remains unrouted and the transaction must eventually abort or install a replacement handler. Frames are authenticated by the hash field, which chains each frame to its computational history: a resumed coroutine can verify that its frame has not been tampered with by checking the hash against the preceding frame. In a deployment, the hash is checked by the IVC circuit (Layer 3) rather than by the ledger model itself; the formalization treats frame integrity as guaranteed by proof verification (section 4.2).

Modeling assumptions. The formal model makes several assumptions that should be stated explicitly. Cryptographic hash collision resistance is assumed: the frame hash chaining is secure under standard assumptions. ZK proof verification is treated as a sound oracle: the model does not formalize the internal structure of IVC proofs but assumes that verified proofs correctly attest to computation integrity. Handler dispatch uses top-of-stack semantics; an empty stack means the effect is unroutable. Frame and UTxO datum sizes are bounded by deployment transaction size limits (not formalized). All identifier domains are drawn from $\mathbb{N}$; the formalization does not enforce type-level distinction between different ID kinds (e.g., *UTXOId* vs. *TxId*); see section 9 for planned improvements.

Adversary model and trust boundaries. The formal guarantees rest on four assumptions, stated here for auditability. *Security Assumption 1 (ZK verifier soundness):* the predicate `allProofsVerified` is a sound oracle—if it returns true, the attested computation was performed correctly. If this assumption fails, all safety guarantees become vacuous; an adversary can forge proof commitments and commit arbitrary invalid transactions. *Security Assumption 2 (Phase discipline):* the transaction executor enforces the phase-transition ordering (Idle → Reserve → Executing → Committing → Committed). The Lean mechanization proves safety given

this ordering; enforcement is a deployment obligation. *Security Assumption 3 (S-BAC Byzantine tolerance):* each shard has at most $f_s < n_s/3$ Byzantine validators. Under this bound, S-BAC guarantees safety (no invalid commits); a shard exceeding this bound can force aborts of any transaction touching its state, acting as a censorship vector but not violating safety. *Security Assumption 4 (Collision-resistant hashing):* the frame hash chain provides computational binding under standard cryptographic assumptions.

The Lean mechanization proves properties of the *abstract model*: ledger-level operational semantics, conflict serializability, MPST projection, and invariant preservation. A concrete deployment must additionally implement ZK verification (Assumption 1), enforce phase discipline (Assumption 2), maintain shard validator honesty below the BFT bound (Assumption 3), and enforce lock timeouts via runtime policy (section 4.6).

3.2 Coordination Scripts as Global Types

In standard eUTxO, there is no formal language for specifying how multiple UTxOs should interact within a transaction. Developers encode interaction protocols implicitly in validator logic: a validator checks that certain outputs exist or that certain conditions hold, but the overall coordination pattern among multiple validators is never stated as a first-class specification. ICE-UTxO makes these protocols explicit using *coordination scripts*—formal specifications of the allowed interactions among transaction participants.

The conceptual foundation is multiparty session types (MPST), developed in the programming languages community to specify communication protocols among multiple processes [Honda et al.(2016)]. In the MPST framework, a "global type" describes the complete protocol from a bird's-eye view—who sends what to whom, in what order, with what choices. "Projection" then extracts each participant's local view of the protocol. ICE-UTxO adopts this framework but replaces communication channels with UTxO references and message-passing processes with ledger validators and effect handlers.

In a coordination script, each participant has a *role* with a *kind*: utxo, iface, or shard. UTxO roles represent on-chain actors that can read, consume, and produce UTxOs. Interface roles represent off-chain services (like oracles) that can process effects. Shard roles represent coordinators that manage handler installation and uninstallation. This role-kind system enforces a permission discipline ensuring that only appropriate actions are performed by each kind of participant—for example, only a utxo role may consume a UTxO, and only a shard role may install or uninstall a handler.

A script with an empty event set is trivially well-formed and corresponds to a standard eUTxO transaction requiring no coordination—this is one sense in which ICE-UTxO is a conservative extension of eUTxO.

The three definitions below formalize role kinds, actions, and scripts. A concrete example (fig. 3) follows immediately, walking through a liquidation protocol line by line.

Definition 3.8 (Role Kind)**.** *RoleKind* ::= *utxo* | *iface* | *shard.*

Definition 3.9 (Action)**.** *The action grammar labels events in a coordination script:*

$$\begin{aligned} \mathit{Action} ::= {} & r_1 \to r_2 : \texttt{raise}(i, \mathit{tag}) \\ & \mid r_1 \to r_2 : \texttt{resume}(i, \mathit{tag}) \\ & \mid r_1 \to r_2 : \texttt{install}(h) \\ & \mid r_1 \to r_2 : \texttt{uninstall}(i) \\ & \mid r : \texttt{read}(u) \mid r : \texttt{consume}(u) \mid r : \texttt{produce}(u) \\ & \mid r : \texttt{lock}(u) \mid r : \texttt{snapshot}(u) \end{aligned}$$

Actions are either communication actions *(with sender r_1 and receiver r_2) for raise/resume/install/uninstall, or* local actions *for UTxO operations performed by a single role.* (Mechanized:

```
script LiquidationProtocol {
  roles: oracle: iface, borrower: utxo,
         liquidator: utxo, coordinator: shard;
  events:
    e1: coordinator -> oracle : install(priceHandler);
    e2: borrower -> oracle : raise(getPrice, ETH_USD);
    e3: oracle -> borrower : resume(priceResult, 1500);
    e4: borrower : consume(collateralUtxo);
    e5: liquidator : produce(liquidatedUtxo);
    e6: coordinator -> oracle : uninstall(priceHandler);
  constraints:
    e1 < e2; e2 < e3; e3 < e4; e4 < e5;
    e5 < e6; e1 < e6;
}
```

Figure 3: Example coordination script with action grammar.

`inductive Action` in Script.lean; accessor functions for UTxO access, interface usage, and role participation are extended in PTB.lean.)

Definition 3.10 (Script)**.** *A script is a tuple* $S = (roles, roleKind, E, lab, <, \#)$ *where* $roles : \mathcal{P}_{fin}(RoleId)$*,* $roleKind : RoleId \to RoleKind$*,* $E : \mathcal{P}_{fin}(EventId)$*,* $lab : EventId \to Action$*,* $< \subseteq E \times E$ *is a strict partial order, and* $\# \subseteq E \times E$ *is a symmetric, irreflexive conflict relation.*

To make the script notation concrete, consider fig. 3 line by line. The script declares four roles: `oracle` (an interface role representing a price feed service), `borrower` and `liquidator` (UTxO roles representing on-chain participants), and `coordinator` (a shard role managing handler lifecycle). Event e_1 installs a price handler on the oracle interface—this must happen before any price queries. Event e_2 has the borrower raise a `getPrice` effect, requesting the current ETH/USD price from the oracle. Event e_3 has the oracle resume the borrower with the price result. Events e_4 and e_5 perform the actual liquidation: the borrower consumes the collateral UTxO and the liquidator produces a new UTxO with the liquidated assets. Finally, e_6 uninstalls the price handler. The constraints encode the causal ordering: the handler must be installed before any price query ($e_1 < e_2$), the query must complete before the result is available ($e_2 < e_3$), the result must be known before liquidation proceeds ($e_3 < e_4 < e_5$), and the handler must remain installed until after liquidation ($e_5 < e_6$, $e_1 < e_6$).

Definition 3.11 (Well-Formedness)**.** *A script* S *is* well-formed*, written* $WF(S)$*, if all of the following hold:*

1. ***orderDom****: both endpoints of every edge in* $<$ *are in* E*.*
2. ***orderAcyclic****:* $<$ *is acyclic (and therefore irreflexive).*
3. ***conflictDom****: both endpoints of every edge in* $\#$ *are in* E*.*
4. ***conflictIrrefl****:* $\#$ *is irreflexive (*$\neg(e \# e)$ *for all* e*).*
5. ***conflictSymm****:* $\#$ *is symmetric (*$e_1 \# e_2 \implies e_2 \# e_1$*).*
6. ***rolesOK****: all roles referenced in event labels are declared in roles.*
7. ***roleKindOK****: UTxO operations (`read`, `consume`, etc.) are performed only by roles of kind utxo;* `install`/`uninstall` *only by shard roles;* `resume` *(as sender) only by iface roles.*

In the Lean formalization, well-formedness is a conjunction of these seven predicates, all defined in `Script.lean`*.*

3.3 Event-Structure Semantics

A coordination script specifies a partial order of events, not a total order. Some events are causally independent—for example, two UTxOs being updated in parallel by different roles—and can occur in any order or even simultaneously. Event structures, introduced by Winskel [Winskel(1987)], capture this concurrency directly, representing the causal dependencies and mutual exclusions among events without requiring enumeration of all possible interleavings. This is a more natural fit for blockchain transactions than sequence-based models, because transactions inherently involve concurrent participants.

A *configuration* is a "snapshot of progress"—the set of events that have completed so far. It must be consistent (no two conflicting events have both occurred) and down-closed (if an event is done, all its causal prerequisites are done too). Think of it as a valid checkpoint in the protocol execution: at any configuration, the protocol is in a well-defined state from which execution can continue.

An event is "ready to fire" (enabled) when all its causal predecessors have occurred and it does not conflict with anything already done. A *valid trace* is a sequence of events that could be produced by repeatedly firing enabled events, starting from an empty configuration. It represents one possible execution of the protocol. The definitions below are mechanized in `Script.lean`.

Definition 3.12 (Configuration)**.** *A set $C \subseteq E$ is a* configuration *of script S if $C \subseteq S.events$, C is conflict-free, and C is down-closed.*

Definition 3.13 (Enablement)**.** *Event e is* enabled *in configuration C if $e \in S.events$, $e \notin C$, all predecessors are in C, and e does not conflict with any event in C.*

Definition 3.14 (Valid Trace)**.** *A* valid trace *is a sequence $[e_1, \ldots, e_n]$ admitting a chain $\emptyset = C_0 \to C_1 \to \cdots \to C_n$ of configurations where each e_i is enabled in C_{i-1}.*

These definitions are replicated for local scripts (`LocalScript`) with identical structure, ensuring the same semantic framework applies at both global and local levels.

3.4 Projection and Local Conformance

If the coordination script is a stage play's full script, projection extracts a single actor's lines and cues. Each role sees only the events it participates in, preserving the causal ordering between those events but discarding events involving other roles. The projected local script is itself a well-formed event structure (Theorem 3.18), so the same semantic machinery—configurations, enablement, valid traces—applies at both the global and local levels.

Projection matters for sharding: in a sharded deployment, each shard only needs to verify its own roles' behavior. Projection formally justifies this decomposition—if the global protocol is correct, every shard's local view is correct (Theorems 3.19 and 3.20), and conversely, if every shard's local view is correct and they are mutually consistent, the global protocol is correct (Theorem 3.41 in section 3.6).

Projection extracts from a global script the view of a single role. The role-kind system enforces a permission discipline on which actions each kind of role may perform:

Action	`utxo`	`iface`	`shard`
`read`, `consume`, `produce`, `lock`, `snapshot`	yes	no	no
`raise` (sender)	yes	no	no
`resume` (sender)	no	yes	no
`install`, `uninstall` (sender)	no	no	yes
Any action (receiver)	yes	yes	yes

This matrix is enforced by the **WF-RoleKind** predicate (theorem 3.11).

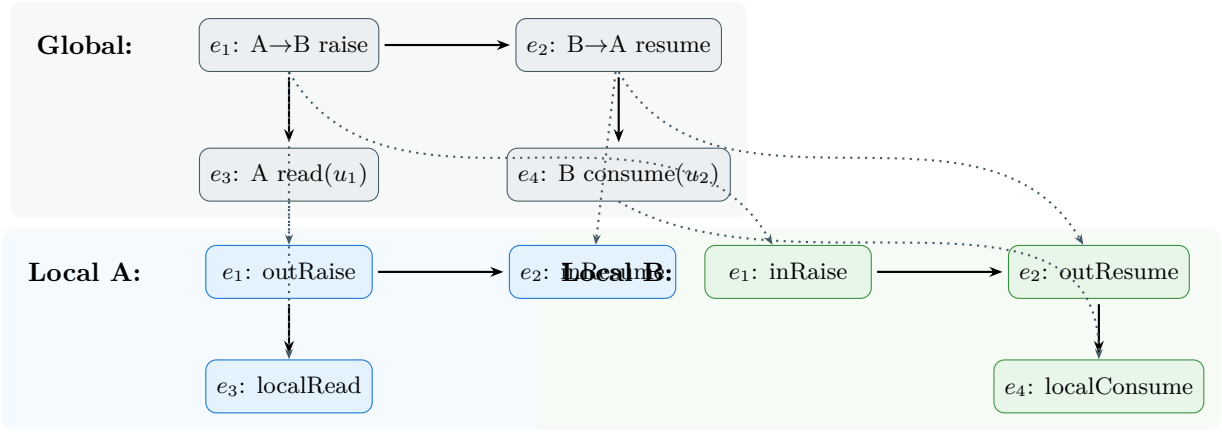

Figure 4: Projection: a global script decomposes into local scripts for each role.

Definition 3.15 (Participation)**.** *Role r* participates *in event e if r appears as sender or receiver in $lab(e)$, or $lab(e)$ is a UTxO action performed by r. Formally, $r \in participants(lab(e))$.*

Definition 3.16 (Relevance)**.** *Event e is* relevant *to role r, written $relevant(r, e)$, if $toLocal(r, lab(e)) \neq None$, where $toLocal$ converts a global action to a local action from r's perspective:*

- $r_1 \to r_2 : \texttt{raise}(i, t)$ *becomes* $\texttt{outRaise}(r_2, i, t)$ *if* $r = r_1$, *or* $\texttt{inRaise}(r_1, i, t)$ *if* $r = r_2$.
- *UTxO actions become local variants (e.g.,* $r : \texttt{read}(u)$ *becomes* $\texttt{localRead}(u)$*).*
- *Otherwise None.*

Definition 3.17 (Projection)**.** *Given script S and role r, the projection $Proj(S, r)$ is the local script $(E_r, lab_r, <_r, \#_r)$ where $E_r = \{e \in E \mid relevant(r, e)\}$ and the relations are restricted to $E_r \times E_r$.*

Theorem 3.18 (Projection Preserves Well-Formedness)**.** *If $WF(S)$ then $WF(Proj(S, r))$ for all roles r.* (Mechanized: `project_wellFormed`, Script.lean.)

Theorem 3.19 (Projection Preserves Traces)**.** *If tr is a valid trace of S, then $traceProj(S, r, tr)$ is a valid trace of $Proj(S, r)$.* (Mechanized: `proj_validTrace`, Script.lean.)

Corollary 3.20 (Global Conformance Implies Local)**.** *If S globally conforms on trace tr, then S locally conforms for every role r on $traceProj(S, r, tr)$.* (Mechanized: `proj_localConform_of_globalConform`, Script.lean.)

Definition 3.21 (Trace Projection)**.** $traceProj(S, r, tr) = tr.\texttt{filter}(relevant(r, \cdot))$.

Definition 3.22 (Local Conformance)**.** *Script S* locally conforms *for role r on trace tr, written $localConform(S, r, tr)$, if $Proj(S, r)$ is well-formed and tr is a valid trace of $Proj(S, r)$.*

3.5 PTB Compilation

A coordination script's event structure admits many valid linearizations—any topological sort of the causal DAG respects the partial order. On-chain validators, however, need a single concrete command sequence to replay and verify. PTB compilation selects one such linearization and encodes it as a Programmable Transaction Block (PTB), inspired by Sui's transaction format [Blackshear et al.(2023)]: a sequence of commands where each command produces a result stored in a numbered register $Result(i)$, and later commands reference earlier results as inputs, creating an explicit dataflow graph.

The key property is determinism. The event structure captures what *could* happen; the PTB captures what *did* happen. This determinism is what makes on-chain validation possible: validators replay the exact command sequence and verify that the result matches the proof. The correctness criterion is that the PTB's induced event structure (derived from its dependency and conflict relations) is well-formed and admits the PTB's command sequence as a valid trace (Theorems 3.32–3.33). A concrete worked example appears in section 3.5.5; the formal definitions follow. All definitions and theorems are mechanized in `PTB.lean`.

3.5.1 Command Model

Definition 3.23 (Command)**.** *A command is a triple $c = (action, uses, conflicts)$ where $action$: Action, $uses$: $\mathcal{P}_{fin}(\mathbb{N})$ identifies result registers consumed as inputs, and $conflicts$: $\mathcal{P}_{fin}(\mathbb{N})$ records explicit conflict edges.*

Definition 3.24 (Program)**.** *A PTB program is a list of commands $P = [c_0, \ldots, c_{n-1}]$. Command c_i stores its output in register Result(i), and may reference Result(j) for $j \in c_i.uses$. Well-formedness requires $j < i$ for every $j \in c_i.uses$.*

Definition 3.25 (Config)**.** *A PTB configuration $(roles, roleKind)$ maps identifiers to roles and roles to kinds, connecting the PTB to the coordination layer's role structure. (In the Lean mechanization, this corresponds to `structure Config` in PTB.lean; an extended variant `AccessConfig` adds utxoRole and ifaceRole mappings used by the witness construction theorems.)*

3.5.2 Derived Relations

Given a program P, we derive three dependency relations (*dataDep*, *utxoDep*, *handlerDep*) that contribute to the causal order, plus one conflict marker (*explicitConflict*) for choice branches, and combine these into two structural relations: *orderRel* (causal order) and *conflictRel* (mutual exclusion). Each action a exposes accessor functions: $a.readUtxos$, $a.writeUtxos$, $a.consumedUtxos$, $a.ifaceUses$, $a.ifaceInstalls$, $a.ifaceUninstalls$.

Definition 3.26 (Dependencies)**.** *For commands c_i, c_j with $i < j$:*

- $dataDep(i, j) \iff i \in c_j.uses$ — *result register dependency.*
- $utxoDep(i, j) \iff c_i.writeUtxos \cap c_j.utxoAccesses \neq \emptyset \vee c_j.writeUtxos \cap c_i.utxoAccesses \neq \emptyset$ — *read-write or write-write overlap.*
- $handlerDep(i, j) \iff c_i.ifaceInstalls \cap c_j.ifaceUses \neq \emptyset \vee c_i.ifaceUses \cap c_j.ifaceUninstalls \neq \emptyset$ — *install-before-use or use-before-uninstall.*

Definition 3.27 (Order Relation)**.** $orderRel(i, j) \iff i < |P| \wedge j < |P| \wedge i < j \wedge (dataDep(i, j) \vee utxoDep(i, j) \vee handlerDep(i, j))$.

Definition 3.28 (Conflict Relation)**.** $conflictRel(i, j) \iff i < |P| \wedge j < |P| \wedge i \neq j \wedge (c_i.consumedUtxos \cap c_j.consumedUtxos \neq \emptyset \vee c_i.ifaceInstalls \cap c_j.ifaceInstalls \neq \emptyset \vee explicitConflict(i, j))$.

Remark 3.29 (Hereditary Conflict)**.** *The relation conflictRel is symmetric (by commutativity of set intersection and the $i \neq j$ guard) and irreflexive (since $i \neq j$ is required), but does* not *enforce hereditary conflict (downward closure of conflict under causality), as required by Winskel's original event structures [Winskel(1987)]. In Winskel's formulation, if $e \# e'$ and $e' \leq e''$, then $e \# e''$: conflict propagates forward through causality. ICE-UTxO omits this because: (i) PTB-derived event structures have finite, acyclic order relations where the precedence graph's acyclicity check (theorem 4.11, component 6) is sufficient for serializability without hereditary closure;*

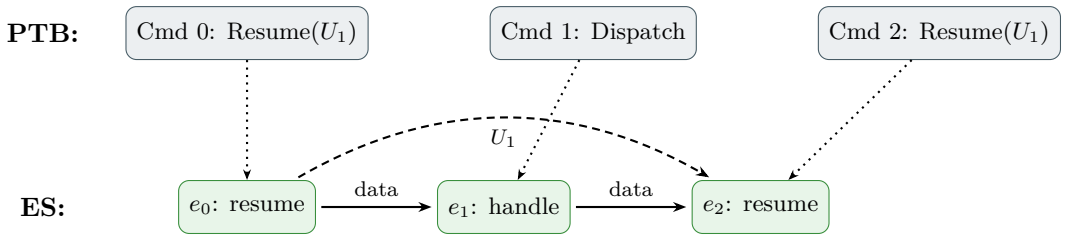

Figure 5: PTB compilation mapping commands to event structures.

(ii) the serializability proof (theorem 5.15) depends on fullConflicts symmetry and CoreState commutativity, not on hereditary propagation; and (iii) the projection theorems (theorems 3.18 and 3.19) hold for the weaker conflict relation. Adding hereditary closure would strengthen the specification against modeling errors in hand-written scripts (where causal chains might introduce implicit conflicts) and is planned (section 9). For PTB-compiled scripts, the omission has no effect on the mechanized results: the toScript translation produces conflict edges only between commands at the same causal depth, so hereditary closure would add no new edges.

3.5.3 Translation to Event Structure

Definition 3.30 (toScript)**.** *Given configuration cfg and program P, the induced script is:*

$$\mathit{toScript}(\mathit{cfg}, P) = (\mathit{cfg.roles}, \mathit{cfg.roleKind}, \{0, \ldots, |P| - 1\}, \lambda i.\, c_i.\mathit{action}, \mathit{orderRel}, \mathit{conflictRel})$$

Remark 3.31 (Compilation Direction)**.** *The translation is deliberately from programs to event structures, not the reverse. The PTB program is the author-facing artifact: it specifies* what to execute *(commands with explicit data and conflict dependencies). The event structure extracted by toScript is the verification artifact: it specifies* what constraints the execution must satisfy *(causal order and conflict). The correctness theorem (theorem 3.32) ensures that the extracted constraints are well-formed. A reverse direction—compiling an arbitrary event structure into a PTB—would require solving a scheduling problem (topological sort with conflict handling and register assignment) and is not needed for the protocol: transaction authors construct PTB programs directly or via a high-level DSL (section 9), and the ledger verifies the induced event structure. The design parallels the relationship between programs and their verification conditions in Hoare logic: the program is primary, the specification is derived.*

Theorem 3.32 (toScript Well-Formedness)**.** *If P satisfies rolesOK and roleKindOK with respect to cfg, then $\mathit{toScript}(\mathit{cfg}, P)$ is well-formed.* (Mechanized: `Program.toScript_wellFormed`, PTB.lean.)

Proof sketch. The seven well-formedness predicates decompose as follows. **WF-Order** and **WF-Conflict**: both endpoints are in $\{0, \ldots, |P| - 1\}$ by the bound guards in the relation definitions. **WF-Irrefl** and **WF-Symm** for conflict: irreflexivity follows from $i \neq j$; symmetry from `conflictRel_symm`. **WF-Acyclic**: if $\mathit{orderRel}^+(i, i)$ held, then $i < i$ by transitivity of the $i < j$ constraint in each order edge—contradiction. **WF-Roles** and **WF-RoleKind**: by hypothesis. □

3.5.4 Valid Traces and Witness Construction

Theorem 3.33 (validTrace_trace)**.** *If P has no conflicts, then the full program-order trace $[0, \ldots, |P| - 1]$ is a valid trace of $\mathit{toScript}(\mathit{cfg}, P)$.* (Mechanized: `Program.validTrace_trace`, PTB.lean.)

Theorem 3.34 (witnessGlobalOK_of)**.** *If P satisfies rolesOK, roleKindOK, conflictFree(keep), and downClosed(keep), then toWitness(cfg, P, keep) is globally OK.* (Mechanized: `Program.witnessGlobalOK_`
PTB.lean.)

Theorem 3.35 (validTrace_traceOf)**.** *Given a predicate keep : $\mathbb{N} \to$ Bool such that P is conflict-free on keep and keep is down-closed with respect to orderRel, the filtered trace P.traceOf(keep) is a valid trace of toScript(cfg, P).* (Mechanized: `Program.validTrace_traceOf`, PTB.lean.)

Theorem 3.36 (crossRoleSafe_of_access)**.** *If every command's access roles are contained in its participant set (accessRolesOK) and explicit conflicts are shared (explicitConflictShared), then conflicting commands always share at least one role.* (Mechanized: `Program.crossRoleSafe_of_access`, PTB.lean.)

Choice and the `keep` predicate. The predicate $keep : \mathbb{N} \to Bool$ selects which commands in a PTB are executed in a given run, modeling branching and conditional logic. For a linear protocol (no choices), $keep = \lambda i.\, true$. For a protocol with an exclusive choice between commands i and j (expressed via $conflictRel(i, j)$), exactly one of $keep(i)$ or $keep(j)$ is true. The two well-formedness conditions—*keep* is conflict-free (no two kept commands conflict) and down-closed (if command j is kept and i *orderRel* j, then i is kept)—ensure that the selected subset forms a valid configuration of the induced event structure (theorem 3.12). The transaction author constructs *keep* during execution (it records which branches were taken) and includes it in the coordination witness. The IVC proof then certifies that the selected commands were executed correctly, binding *keep* into the *commitHash* so the ledger cannot be tricked into accepting an inconsistent subset.

3.5.5 Worked Example

Consider the collateralized loan scenario from section 1.1, compiled to a 3-command PTB:

i	Command	Action	uses	conflicts
0	Resume borrower coroutine	`resume`$(r_b, r_o, oracle, getPrice)$	$\emptyset$	$\emptyset$
1	Dispatch to price handler	`raise`$(r_o, r_h, oracle, dispatch)$	$\{0\}$	$\emptyset$
2	Resume with price result	`resume`$(r_h, r_b, oracle, result)$	$\{1\}$	$\emptyset$

Derived edges. $dataDep(0, 1)$ since $0 \in c_1.uses$; $dataDep(1, 2)$ since $1 \in c_2.uses$; $handlerDep(0, 2)$ since both access the oracle interface. Thus $orderRel = \{(0, 1), (1, 2), (0, 2)\}$ and $conflictRel = \emptyset$.

Induced script. *toScript* produces events $\{0, 1, 2\}$ with order $0 < 1 < 2$ (plus $0 < 2$) and no conflict. This is a total order, so the unique valid trace is $[0, 1, 2]$.

Validity. Since $conflictRel = \emptyset$, theorem 3.33 applies directly: the program-order trace $[0, 1, 2]$ is valid. By theorem 3.34, $toWitness(cfg, P, \lambda i.\, true)$ is globally OK.

Step-by-step configuration sequence. We can trace the execution through the formal definitions. Starting from the empty configuration $C_0 = \emptyset$: event 0 (resume borrower) is enabled in C_0 because it has no predecessors and no conflicts, so $C_1 = \{0\}$. Event 1 (dispatch to handler) is enabled in C_1 because its sole predecessor 0 is present, so $C_2 = \{0, 1\}$. Event 2 (resume with result) is enabled in C_2 because predecessors 0 and 1 are both present, so $C_3 = \{0, 1, 2\}$. The configuration C_3 is maximal (all events have occurred), confirming that the trace $[0, 1, 2]$ is complete.

Connection to the witness. The valid trace $[0, 1, 2]$ is exactly the evidence packaged into the coordination witness $W = (S, [0, 1, 2])$, which the IVC proof then certifies. A verifier need not re-execute the three commands; it checks that W satisfies *witnessGlobalOK* (which reduces to checking that $[0, 1, 2]$ is a valid trace of the induced script S) and that the IVC proof attests to the computation's correctness.

What the IVC proof certifies. The transaction author generates an IVC proof certifying: (i) the borrower's coroutine correctly computed the loan terms from the price data (execution correctness); (ii) the price-oracle effect was dispatched to the correct handler and the result propagated (effect resolution); and (iii) the witness $W = (S, [0, 1, 2])$ satisfies *witnessGlobalOK* (protocol conformance). The *commitHash* binds this proof to the specific input UTxOs (borrower's collateral, oracle's price feed) and output UTxOs (the loan position, updated collateral). A validator checks the IVC proof and the structural predicates (*commitEnabledStrong*) without re-executing the coroutine logic.

Shard-local verification. Suppose the borrower resides on shard 1 and the oracle on shard 2. Under S-BAC (section 3.6), shard 1 checks: (i) *witnessLocalOK* for the borrower role r_b—that the projected trace $[0, 2]$ locally conforms to $Proj(S, r_b)$; and (ii) that the borrower's input UTxOs are live. Shard 2 checks the same for oracle role r_o with projected trace $[0, 1]$. By theorem 3.40, these local checks are implied by the global witness validity; by theorem 3.41, their conjunction (plus trace consistency) recovers the global guarantee. Neither shard needs to see the other's UTxO set or re-execute the other's coroutine steps.

3.6 S-BAC Integration

When a transaction's participants span multiple shards—for example, a borrower on shard 1 and an oracle on shard 2—the transaction cannot be validated by a single shard. The shards must synchronize to ensure atomicity: either all shards commit, or all abort. This is the classic atomic commit problem, complicated by the possibility of Byzantine (adversarial) faults.

Sharded Byzantine Atomic Commit (S-BAC), introduced by Chainspace [Al-Bassam et al.(2018)], addresses this with a two-phase protocol. In the *prepare* phase, each shard independently checks its local portion of the transaction (local trace conformance for its roles and UTxO liveness for its inputs). In the *accept* phase, if all shards prepared successfully, the transaction commits; if any shard aborts, all abort. This is similar to classical two-phase commit (2PC) but tolerant of Byzantine faults within each shard's BFT consensus.

ICE-UTxO improves over Chainspace's original design in one key respect: in Chainspace, each shard re-executes the transaction's logic during prepare. In ICE-UTxO, shards only check their local trace conformance (is my role following the protocol?) and UTxO liveness (are the inputs still unspent?). The heavy computation—verifying that the full interleaving respects the coordination script—is certified by the IVC proof, which shards verify rather than re-execute. This reduces both the computation overhead per shard and the amount of private data that must be revealed.

The definitions below formalize how coordination witnesses connect to the S-BAC commit decision.

Definition 3.37 (S-BAC Configuration)**.** *An S-BAC configuration is a pair* (*shardOfUtxo*, *rolesOfShard*) *mapping UTxOs to shards and shards to their associated roles.*

Definition 3.38 (Coordination Witness)**.** *A coordination witness is a pair* $W = (S, tr)$ *of a script* S *and a trace* tr*.*

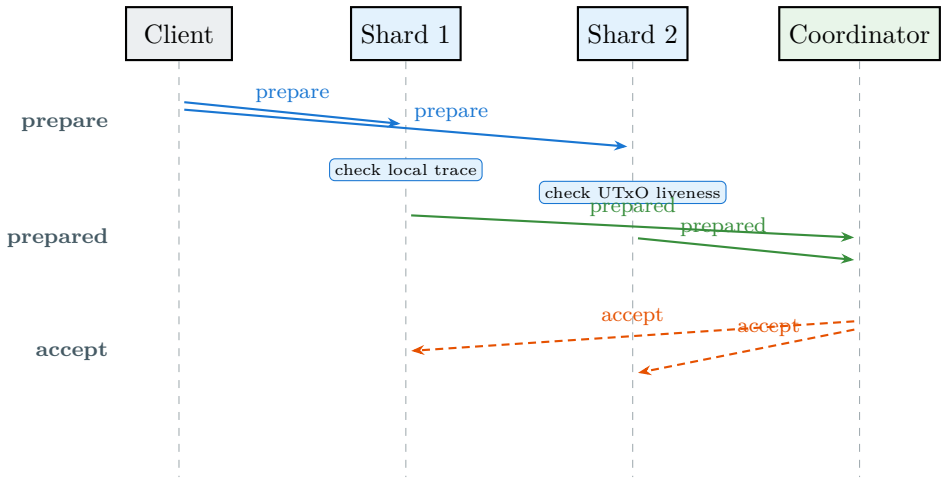

Figure 6: S-BAC prepare flow for ICE-UTxO. Dashed vertical lines are lifelines; three temporal phases flow top-to-bottom.

Definition 3.39 (Witness Predicates)**.**

$$\begin{aligned} \mathit{witnessGlobalOK}(W) &\iff S.\mathit{globalConform}(tr) \\ \mathit{witnessLocalOK}(W) &\iff \forall r \in S.\mathit{roles}.\ S.\mathit{localConform}(r, S.\mathit{traceProj}(r, tr)) \\ \mathit{witnessConsistent}(W) &\iff S.\mathit{traceConsistent}(tr) \end{aligned}$$

Theorem 3.40 (Global Implies Local)**.** $\mathit{witnessGlobalOK}(W) \implies \mathit{witnessLocalOK}(W)$. (Mechanized: `witnessLocalOK_of_global`, SBAC.lean.)

Theorem 3.41 (Consistent Implies Global)**.** *If S is well-formed and $\mathit{witnessConsistent}(W)$, then $\mathit{witnessGlobalOK}(W)$.* (Mechanized: `witnessGlobalOK_of_local_and_consistent`, SBAC.lean.)

Definition 3.42 (Coordination Transaction)**.** *A coordination transaction pairs a ledger transaction with a witness: $ctx = (tx, W)$.*

Definition 3.43 (Coordinated Commit)**.** *Commit is enabled if the ledger commit guard passes* and *the witness validates:*

$$\mathit{coordCommitEnabled}(m, L, ctx) \iff \mathit{commitEnabledStrong}(m, L, tx) \wedge \mathit{witnessGlobalOK}(W)$$

Definition 3.44 (Shard-Local Check)**.** *During S-BAC prepare, shard s checks:*

1. ***Local trace conformance**: for every role r associated with s, $\mathit{localConform}(S, r, \mathit{traceProj}(S, r, tr))$.*
2. ***UTxO liveness**: inputs assigned to shard s are in $L.utxos$.*

Theorem 3.45 (Local Sufficiency)**.** *If $\mathit{coordCommitEnabled}(m, L, ctx)$ holds, then $\mathit{coordCommitEnabledLocal}(n$ holds; that is, shard-local checks are sufficient when the global witness is valid.* (Mechanized: `coordCommitEnabledLocal_of_global`, SBAC.lean.)

The definitions and theorems in this section establish *safety* properties: the model specifies what constitutes a valid execution and proves that the commit mechanism admits only valid executions. Conditional *liveness* properties—guaranteeing that well-formed protocols eventually complete under fairness assumptions—are established in section 4.6 using TLA-style reasoning with mechanized support lemmas and TLC model-checking validation.

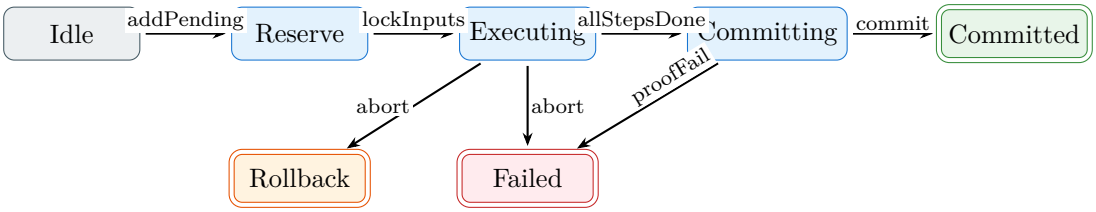

Figure 7: Transaction lifecycle state machine.

Table 4: Step constructors of the ICE-UTxO operational semantics.

Constructor	Mode	Precondition	Effect	Intuition
`addPending`	any	$tx \notin L.pending$, `noDupProofs`	Add to pending	Submit for consideration
`lockInputs`	locking	$tx \in pending$, inputs disjoint, live	Lock inputs	Reserve inputs (pessimistic)
`commit`	any	`commitEnabledStrong`, acyclicity	Apply commit	Finalize—strongest gate
`abort`	any	$tx \in pending$	Release locks	Graceful failure / timeout
`installH`	any	(none)	Push handler	Register effect handler
`uninstallH`	any	(none)	Pop handler	Deregister effect handler
`raiseE`	any	(none)	Push effect	Request cross-boundary service
`handleE`	any	effect + handler present	Route effect	Dispatch to registered handler

4 Operational Semantics and Ledger Safety

The preceding sections defined *what* ICE-UTxO transactions look like—state components, coordination scripts, event structures, and compiled PTB programs. This section defines *how* those transactions change the ledger: a small-step operational semantics with eight constructors, two concurrency modes, and six invariants preserved by every step. All definitions and theorems are mechanized in `StarstreamPilot.lean`.

4.1 Transaction Lifecycle

Running example. Consider the collateralized loan liquidation from section 1.1 and its PTB program from section 3.5.5. The transaction would progress through the lifecycle below as follows: submit to pending (`addPending`), acquire locks on the oracle and borrower UTxOs (`lockInputs`), install the price handler (`installH`), raise the price query (`raiseE`), dispatch the handler (`handleE`), and finally commit with the IVC proof (`commit`).

A transaction proceeds through seven phases: Idle, Reserve, Executing, Committing, Committed, Rollback, and Failed.

Definition 4.1 (Step)**.** *The small-step relation* $Step : Mode \to Ledger \to Ledger \to Prop$ *has eight constructors in three groups:* lifecycle *(addPending, lockInputs, commit, abort) governs transaction admission, resource reservation, finalization, and failure;* effect *(installH, uninstallH, raiseE, handleE) implements the algebraic-effects paradigm at the ledger level; and the lifecycle constructors carry the mode-sensitive preconditions that enforce concurrency control.*

Entry gate: `addPending`. The `addPending` constructor admits transactions to the pending set. Its precondition—$tx \notin L.pending$ and `noDupProofs`—is deliberately minimal: any structurally well-formed transaction passes. No input liveness, proof validity, or phase correctness is checked here. All semantic validation is deferred to later lifecycle phases, keeping admission lightweight.

Remark 4.2 (Security: Spam and DoS Surface)**.** *The* ***`addPending`*** *constructor admits any structurally well-formed transaction without checking proof validity or input liveness. This*

makes it the primary denial-of-service attack surface: an adversary can flood the pending set with transactions that will never commit. Mitigation via fees, rate limiting, or proof-of-work puzzles is essential in deployment but outside the model's scope.

Pessimistic reserve: `lockInputs`. The `lockInputs` constructor implements pessimistic concurrency control, reserving a transaction's inputs before execution begins. The preconditions—inputs disjoint from already-locked UTxOs and inputs live—mirror the lock-acquisition phase of strict two-phase locking (S2PL) in database systems. In the loan example, the borrower's collateral UTxO and the oracle's price-feed UTxO are locked before the price query is dispatched, ensuring that no concurrent transaction can consume them while the multi-step interaction is in progress.

Terminal gate: `commit`. This is the hardest gate to pass. The `commit` constructor carries the strongest preconditions of any constructor. It requires: (i) all proof commitments verified (*allProofsVerified*), (ii) structural validity (*validTx*), (iii) the transaction is in the committing phase, (iv) the transaction is not already in the history, and (v) both acyclicity predicates hold after appending. Each guard prevents a specific failure mode: (i) blocks unverified computation, (ii) blocks double-spends and stale inputs, (iii) enforces lifecycle discipline, (iv) prevents duplicate commits, and (v) ensures the resulting history admits a conflict-serializable schedule (section 5).

Graceful failure: `abort`. The `abort` constructor is intentionally nondeterministic: any pending transaction may abort at any time, modeling timeouts, cancellation, and resource limits. The safety philosophy is asymmetric: safety is not "valid transactions must commit" but "only valid transactions *can* commit." Making abort unconditional ensures that the model's safety proofs hold regardless of the abort policy a deployment chooses—whether time-based, fee-based, or operator-initiated.

Effect quartet: `installH`, `uninstallH`, `raiseE`, `handleE`. These four constructors implement the algebraic-effects paradigm at the ledger level. In the loan example, the sequence is: (1) `installH` registers a price-oracle handler on the oracle interface; (2) `raiseE` raises a "get price" effect on that interface; (3) `handleE` dispatches the effect to the installed handler, which produces the price data; (4) optionally, `uninstallH` removes the handler after use. The constructors have no preconditions restricting their interleaving. In particular, an `uninstallH` step can remove a handler while a `raiseE` step targeting the same interface is pending. The model handles this by design: `handleE` requires both a pending effect and a handler to be present, so an uninstall-before-handle sequence simply leaves the effect unrouted, and the transaction must eventually abort or install a replacement handler.

Remark 4.3 (Handler Stack Discipline)**.** *The handler stack follows a LIFO (last-in, first-out) discipline: the most recently installed handler for an interface is the one that receives the next effect raised on that interface. This mirrors the handler-stack semantics of algebraic effect systems in programming languages (e.g., Eff, Koka, OCaml 5). The LIFO ordering interacts safely with concurrent uninstallation: if handler h_1 is installed, then h_2 is installed on the same interface, and h_1 is subsequently uninstalled, the stack pops h_1 but h_2 remains and continues to service effects. If* all *handlers for an interface are uninstalled while effects are pending, the effects remain in the queue indefinitely; the transaction cannot reach the committing phase (since unrouted effects imply incomplete execution) and must eventually abort under $WF(abort)$ (section 4.6).*

Remark 4.4 (Effects and Commit Admissibility)**.** *The commit guard commitEnabledStrong (theorem 4.9) does not explicitly check for unresolved effects. Instead, effect resolution is enforced*

through the IVC proof. *The IVC proof predicate (theorem 3.5) includes effect resolution as a subgoal: a valid proof certifies that every raised effect was dispatched and handled. A transaction with unresolved effects cannot produce a valid IVC proof, so allProofsVerified will not hold, and the transaction cannot commit. This design keeps the commit guard lean (it checks a single cryptographic predicate rather than enumerating execution-level conditions) and places the burden of proving completeness on the transaction author's prover rather than on the on-chain validator. The connection is conditional on Security Assumption 1 (ZK verifier soundness): if the verifier is unsound, a transaction with unresolved effects could forge a proof and commit.*

Definition 4.5 (State Updates)**.**

$$\begin{aligned} \mathit{applyCommit}(L, tx) = L\Big[& \mathit{utxos} := (L.\mathit{utxos} \setminus tx.\mathit{inputs}) \cup tx.\mathit{outputs}, \\ & \mathit{consumed} := L.\mathit{consumed} \cup tx.\mathit{inputs}, \\ & \mathit{locked} := L.\mathit{locked} \setminus tx.\mathit{inputs}, \\ & \mathit{pending} := L.\mathit{pending} \setminus \{tx\}, \\ & \mathit{history} := L.\mathit{history} +\!\!+ [tx]\Big] \end{aligned}$$

$$\mathit{applyAbort}(L, tx) = L\Big[\mathit{locked} := L.\mathit{locked} \setminus tx.\mathit{inputs},\ \mathit{pending} := L.\mathit{pending} \setminus \{tx\}\Big]$$

Definition 4.6 (Multi-Step)**.** *Steps : Mode → Ledger → Ledger → Prop is the reflexive-transitive closure of Step.*

4.2 Commit Rule and Proof-Gating

The commit rule enforces that only proof-verified, structurally valid transactions can extend the ledger history.

Definition 4.7 (All Proofs Verified)**.** $\mathit{allProofsVerified}(tx) \iff tx.\mathit{proofCommitments} \neq [] \wedge \forall p \in tx.\mathit{proofCommitments}.\ p.\mathit{phase} = \mathit{Verified}$

Trust boundary. The predicate *allProofsVerified* checks phase flags set by an external ZK verifier. Cryptographic soundness—that a `Verified` flag genuinely corresponds to a valid ZK proof—is outside the scope of this formalization. The model proves: *if* the verifier sets flags correctly, *then* the ledger maintains its invariants.

IVC proof failure and timeout. A proof commitment that fails during generation or verification transitions to the `Failed` phase (theorem 3.4). Since *allProofsVerified* requires *every* commitment to be in the `Verified` phase, any single proof failure prevents the transaction from committing. The transaction remains in the pending set until it is aborted. There is no partial-proof recovery: if k of n proofs verify but the remaining fail, the entire transaction must abort and retry.

Security assumptions. The formal guarantees rest on three explicit assumptions:

1. **ZK verifier soundness.** The external ZK verification oracle is assumed sound: a proof commitment reaches `Verified` only if the underlying computation genuinely satisfies the stated predicate. If violated, all safety guarantees collapse.
2. **Phase-transition integrity.** The `Step` constructors do not enforce that transaction phases progress in the correct order. In a deployment, the transaction executor is responsible for enforcing the intended phase discipline. Enforcing phase progression at the type level is left to future work (section 9).

3. **S-BAC shard honesty.** The cross-shard atomic commit protocol (section 3.6) assumes $f_s < n_s/3$ Byzantine validators per shard, consistent with the Chainspace fault model [Al-Bassam et al.(2018)].

Phase ordering in practice. Assumption (2) deserves elaboration. The `Step` relation is intentionally permissive: it does not constrain the order in which constructors are applied. For instance, nothing in the model prevents a `commit` step from being attempted before `lockInputs`—the preconditions will simply fail. This design keeps the formal model simple and makes the safety proofs independent of any particular scheduling strategy. However, a deployment must enforce the intended lifecycle (fig. 7) externally—for example, by wrapping the step constructors in a state-machine executor that rejects out-of-order transitions, or by encoding the lifecycle phases into the transaction type at the language level. The key guarantee is that even if a buggy executor applies steps in the wrong order, no invalid transaction can commit: the `commitEnabledStrong` guard (theorem 4.9) is the sole gateway. The safety proofs are therefore strictly stronger than a phase-restricted model would provide: they hold for the full state space, including all ill-phased interleavings, so any deployment that restricts to well-phased executions inherits the guarantees automatically as a corollary.

Security considerations. Beyond the three assumptions above, the operational semantics expose several attack surfaces that a deployment must address:

1. **Griefing via lock exhaustion.** In locking mode, an adversary can submit transactions that lock high-demand UTxOs and then stall indefinitely (never committing or aborting). The model permits this because the `abort` constructor is nondeterministic—it does not fire automatically. A deployment must enforce lock timeouts (section 4.4) so that stalled locks are eventually released.

2. **Front-running in optimistic mode.** An adversary who observes a pending transaction's read set can race to commit a conflicting transaction that invalidates the victim's snapshot, forcing an abort. Repeated front-running can starve a target transaction indefinitely. Mitigation strategies include commit-ordering fairness policies, encrypted mempools, or falling back to locking mode for high-value transactions.

3. **Handler manipulation.** An adversary who can issue `installH` for an interface before a legitimate `raiseE` arrives can intercept the effect and route it to a malicious handler. The model places no restriction on who may install handlers; in a deployment, contract authors must restrict handler installation—e.g., to the roles designated in the coordination script (section 6.2). The coordination script provides the formal mechanism for this restriction: the global type assigns roles to interfaces, and the projected local scripts specify which roles may install handlers on which interfaces. A shard checking *witnessLocalOK* for its roles will reject a trace in which an unauthorized role installs a handler, because the projected trace will not conform to the local script. Handler authorization is thus enforced by the same witness-verification mechanism that enforces protocol compliance.

4. **Proof forgery.** If the ZK verifier soundness assumption is violated (e.g., due to a bug in the proving circuit or a break in the underlying cryptographic primitive), an adversary can forge proof commitments and commit invalid transactions. All six ledger invariant components depend on proof-gating (theorem 4.10); a forged proof bypasses the only gate. Defense in depth—such as redundant verification by multiple independent provers—can mitigate this risk.

On-chain verification. A validator (or shard) processing a transaction checks three things: (i) the IVC proof verifies under the stated *verifyKey*, attesting to execution correctness, effect resolution, and protocol conformance (theorem 3.5); (ii) *commitEnabledStrong* holds—inputs are live, outputs are fresh, the transaction is in the committing phase, and mode-specific conditions are satisfied (theorem 4.9); and (iii) the coordination witness satisfies *witnessGlobalOK*, decomposed to shard-local checks via theorem 3.40. Validators do *not* re-execute coroutine logic, replay effect dispatch, or re-check the coordination protocol—all of this is covered by the IVC proof. The on-chain cost is dominated by proof verification (typically under 10ms for a Groth16 or Halo2 proof) plus the *commitEnabledStrong* structural checks (set membership tests on UTxO identifiers).

Definition 4.8 (Valid Transaction)**.** *validTx*(L, tx) *requires:*

- $tx.inputs \neq \emptyset$ *and* $tx.outputs \neq \emptyset$
- $tx.inputs \cap tx.outputs = \emptyset$ *(inputs disjoint from outputs)*
- $tx.inputs \subseteq L.utxos$ *(inputs are live)*
- $tx.outputs \cap (L.utxos \cup L.consumed) = \emptyset$ *(outputs are fresh)*

Definition 4.9 (Commit Enabled—Strong)**.** *commitEnabledStrong*(m, L, tx) *holds if:*

$$allProofsVerified(tx) \wedge validTx(L, tx) \wedge tx.phase = Committing \wedge tx \notin L.history$$

and additionally:

- ***Locking mode****:* $tx.inputs \subseteq L.locked$ *and* $tx.readSet \subseteq L.utxos$*.*
- ***Optimistic mode****:* $tx.readSet \subseteq L.utxos$ *and* $tx.outputs$ *are fresh.*

Theorem 4.10 (Proof-Gated Commit)**.** *Any step that extends the history requires proof verification:*

$$Step(m, L, L') \wedge L'.history = L.history +\!\!+ [tx] \implies proofOk(tx)$$

(Mechanized: `commit_requires_proof`, StarstreamPilot.lean.)

Proof sketch. Exhaustive case analysis on the eight `Step` constructors. Seven non-commit constructors leave the history unchanged, yielding a contradiction with the hypothesis via the lemma $xs \neq xs +\!\!+ [x]$. The commit constructor directly provides *allProofsVerified*(tx) from *commitEnabledStrong*. □

4.3 Concurrency Modes

ICE-UTxO supports two concurrency modes, selected per-transaction:

Locking mode ($m =$ locking). Inputs are reserved before execution via the `lockInputs` step, which requires $tx.inputs \cap L.locked = \emptyset$ (no double-locking) and $tx.inputs \subseteq L.utxos$ (inputs live). At commit time, the commit guard checks $tx.inputs \subseteq L.locked$.

Optimistic mode ($m =$ optimistic). No locks are acquired. At commit time, the guard checks that the read snapshot is still valid ($tx.readSet \subseteq L.utxos$) and outputs are still fresh. If the snapshot has been invalidated by a concurrent commit, the transaction fails.

Both modes are proof-gated: *allProofsVerified* is required regardless of the concurrency mode.

Adversarial contention in optimistic mode. Under adversarial contention, the optimistic mode abort rate can approach 100%: an adversary who can commit transactions consuming UTxOs in a target transaction's read set can force repeated snapshot invalidation. The safety guarantee is that *only valid transactions can commit*, not that valid transactions *will* commit. Liveness under contention requires fairness assumptions (section 4.6).

Comparative analysis. ICE-UTxO's dual-mode concurrency occupies a distinctive point in the blockchain design space. Ethereum executes transactions sequentially under a global state lock: every transaction sees the full state after the previous one, eliminating read-set conflicts at the cost of zero intra-block parallelism. Sui [Blackshear et al.(2023)] achieves parallelism through object-based ownership—non-conflicting transactions on disjoint object sets execute concurrently—but requires Move's linear type system to enforce ownership statically. Cosmos SDK chains [Kwon and Buchman(2016)] isolate state per application chain, eliminating cross-contract conflicts within a chain but requiring IBC for cross-chain coordination. ICE-UTxO differs from all three: it offers *per-transaction mode selection* (locking or optimistic) within a single ledger, gates every commit on a verified proof artifact, and uses algebraic effects to coordinate multi-step interactions—combining the parallelism benefits of UTxO-based systems with the expressiveness typically associated with account-model systems.

4.4 Ledger Invariants

Definition 4.11 (Ledger Invariant)**.** *ledgerInvariant*(L) *is the conjunction of: (1) no double-spend; (2) locked subset active; (3) history nodup; (4) committed implies verified; (5) extended precedence acyclic; (6) full precedence acyclic.*

Theorem 4.12 (Invariant Preservation)**.** *Every step preserves the ledger invariant:*

$$\mathit{Step}(m, L, L') \wedge \mathit{ledgerInvariant}(L) \implies \mathit{ledgerInvariant}(L')$$

(Mechanized: `step_preserves_invariant`, StarstreamPilot.lean.)

Proof sketch. Floyd-Hoare style case analysis on the eight `Step` constructors. `addPending`, `installH`, `uninstallH`, `raiseE`, `handleE` do not modify *utxos*, *consumed*, or *history*, so all six invariant components are trivially preserved. `lockInputs`: *locked* grows by $tx.\mathit{inputs}$, which are live ($\subseteq L.\mathit{utxos}$), preserving `lockedSubsetActive`. `abort`: *locked* shrinks. `commit`: no-double-spend follows from `commit_preserves_no_double_spend`; history nodup from the freshness guard $tx \notin L.\mathit{history}$; acyclicity from the commit step's preconditions. □

Lock lifetime and deadlock prevention. The formalization imposes no lock expiration: locks persist until the holding transaction commits or aborts. This is deliberate—the model captures the worst case (unbounded lock duration) so that safety proofs do not depend on timing assumptions. In practice, a deployment must enforce lock timeouts to prevent indefinite lock holding; the key observation is that any timeout policy can be implemented as an external abort trigger, and the model's safety properties are preserved under *any* abort policy, since the `abort` constructor is unconditionally available (theorem 4.15).

Deadlock among locking-mode transactions cannot arise under a simple discipline: if all transactions acquire locks in a fixed total order over UTxO identifiers, the resulting lock-acquisition graph is acyclic. The `lockInputs` precondition ($tx.\mathit{inputs} \cap L.\mathit{locked} = \emptyset$) already prevents a transaction from acquiring a lock held by another, which forces the failing transaction to abort and retry—effectively converting potential deadlocks into aborts. The unconditional abort constructor thus serves as a safety valve: any transaction that cannot acquire its locks will eventually abort under $WF(\text{abort})$ (section 4.6), releasing its held locks for others.

Theorem 4.13 (No Double-Spend Preservation)**.** $noDoubleSpend(L) \land outputsFresh(L, tx) \land inputsLive(L, tx) \implies noDoubleSpend(applyCommit(L, tx))$. (Mechanized: `commit_preserves_no_double_s`

Theorem 4.14 (Consumed Monotonicity)**.** $Step(m, L, L') \implies L.consumed \subseteq L'.consumed$. *This extends to multi-step:* $Steps(m, L_0, L_n) \implies L_0.consumed \subseteq L_n.consumed$. (Mechanized: `consumed_monotone_step`, `consumed_monotone_steps`.)

Theorem 4.15 (Progress)**.** *Any pending transaction can take a step (at minimum via abort):* $tx \in L.pending \implies \exists L'.\ Step(m, L, L')$. (Mechanized: `pending_can_step`.)

4.5 Reduction to eUTxO

Proposition 4.16 (Conservative Extension)**.** *There exists a structure-preserving embedding* $\iota : Ledger_{eUTxO} \to Ledger_{ICE}$ *such that for every eUTxO step* $L \xrightarrow{s} L'$*, the lifted step* $\iota(L) \xrightarrow{s'} \iota(L')$ *is a valid ICE-UTxO step, and conversely, every ICE-UTxO safety property (no double-spend, input liveness, output freshness) restricted to the image of* ι *coincides with the corresponding eUTxO property.*

Proof sketch. The embedding ι maps an eUTxO ledger state to an ICE-UTxO state with empty handler stacks, empty effect queues, and trivial (single-step) proof commitments. Under this embedding:

- The handler stacks and effect queues remain empty throughout execution, so the effect constructors (`installH`, `uninstallH`, `raiseE`, `handleE`) are never invoked.
- Each input UTxO is validated exactly once (no resume/yield cycle), so the coroutine machinery is inert.
- The transaction applies atomically via a single `commit` step, since no coordination is needed.
- The proof commitment degenerates to a trivial witness (single-step IVC proof), satisfying *allProofsVerified* vacuously.

Safety properties transfer directly: no-double-spend, consumed monotonicity, and the full ledger invariant hold by theorems 4.12 to 4.14, and their restrictions to $img(\iota)$ are exactly the standard eUTxO properties. This argument is a proof sketch, not a mechanized result; the embedding ι and the step-lifting have not been formalized in Lean. The individual safety theorems it invokes *are* mechanized. □

Corollary 4.17. *For transactions with empty handler stacks, empty effect queues, singleton (trivially-verified) proof commitments, and no coordination witness, the commitEnabledStrong predicate reduces to: inputs live, outputs fresh, inputs disjoint from outputs, and proof verified. These are exactly the standard eUTxO validation conditions. The coordination, coroutine, and effect machinery is inert: no* `installH`*,* `raiseE`*,* `handleE`*, or* `uninstallH` *steps are needed, and the commit step applies atomically as in standard eUTxO.*

4.6 Conditional Liveness

The safety properties above guarantee that only valid transactions can commit, but they do not guarantee that valid transactions *eventually do* commit. This section establishes conditional liveness properties under explicit fairness assumptions. The supporting lemmas are mechanized in Lean 4; the liveness theorems themselves are paper-level arguments grounded in these lemmas and validated by TLC model checking.

4.6.1 Fairness Assumptions

We adopt Lamport's temporal logic of actions (TLA) [Lamport(1994)] to state fairness conditions. For an action A:

- **Weak fairness** $WF(A)$: if A is continuously enabled, it eventually fires. Formally, $\Box(\Box\, ENABLED(A) \Rightarrow \Diamond A)$.
- **Strong fairness** $SF(A)$: if A is infinitely often enabled, it eventually fires. Formally, $\Box(\Box\Diamond\, ENABLED(A) \Rightarrow \Diamond A)$.

We assume: (1) WF(commit): if commit is continuously enabled, the scheduler eventually executes it. (2) WF(abort): if a transaction remains pending, the scheduler eventually aborts it. (3) WF(handleE): if effect handling is continuously enabled, the scheduler eventually dispatches it.

Deployment interpretation. Each fairness assumption corresponds to a concrete operational requirement. WF(commit) requires that the validator infrastructure does not indefinitely stall proof-verified transactions—in practice, this is enforced by block-production schedules and transaction-inclusion incentives. WF(abort) requires that timeout mechanisms exist: a pending transaction that cannot make progress (e.g., because its proof generation failed or a counterparty disappeared) must eventually be garbage-collected. WF(handleE) requires that the effect-routing subsystem is live—handlers registered for an interface must eventually process pending effects on that interface.

Fairness failure modes. When these assumptions are violated, the system degrades predictably rather than unsafely. If WF(commit) fails, proof-verified transactions accumulate in the pending set without committing; the ledger remains safe (no invalid commits) but throughput drops to zero. If WF(abort) fails, stale transactions hold locks indefinitely, starving other transactions that need the same inputs—a liveness failure that manifests as lock exhaustion in the locking mode. If WF(handleE) fails, unrouted effects accumulate and coroutines stall at yield points, preventing the transactions that raised those effects from reaching the committing phase. In all three cases, safety invariants (theorem 4.12) continue to hold; only liveness degrades.

4.6.2 Ledger-Level Progress (L1, L2)

Theorem 4.18 (Eventual Commit Under Stability (L1))**.** *Under WF(commit), if a transaction tx remains pending and commitEnabledStrong holds continuously (including both acyclicity conditions), then tx eventually commits.*

Proof sketch. By `commit_step_specific` (Lean), the commit step for tx exists in every state where the preconditions hold. Continuous enabledness plus WF(commit) forces the step to fire. By `commit_adds_to_history`, tx enters the history. By `commit_removes_from_pending`, tx leaves the pending set. □

Theorem 4.19 (Eventual Terminalization (L2))**.** *Under WF(abort), every pending transaction eventually either commits or leaves the pending set.*

Proof sketch. By `abort_enabled_of_pending`, abort is always enabled for any pending transaction. Under WF(abort), the abort step eventually fires unless another step removes tx from pending first. By `abort_removes_from_pending`, abort removes tx from the pending set. □

4.6.3 Effect-Handling Progress (L3)

Theorem 4.20 (Effect Handling Progress (L3))**.** *Under WF(handleE) for interface i, if a handler remains installed on i and effects are pending on i, then the effect queue for i eventually empties.*

Proof sketch. By `handleEffect_succeeds`, `handleE` is enabled when both an effect and a handler are present. By `handleEffect_decreases_effects`, each `handleE` step strictly decreases the effect queue length. The queue length is a natural number, so by well-founded induction the queue empties after at most n steps. □

4.6.4 Coordination and Cross-Shard Progress (L4, L5)

Proposition 4.21 (Coordination Completion (L4))**.** *If a PTB program is conflict-free and all roles participate fairly, the induced event-structure trace can be extended to a complete configuration.*

Argument. By `toScript_wellFormed`, the induced event structure has acyclic order. In any well-formed event structure, every non-maximal configuration has an enabled event. Under fair role participation, enabled events are eventually executed. The trace extends until maximal. This follows the global progress argument of Coppo et al. [Coppo et al.(2016)] instantiated in the event-structure setting of Castellani et al. [Castellani et al.(2023)]. □

Remark 4.22 (Cross-Shard Termination (L5))**.** *Under partial synchrony (DLS model) with $f_s < n_s/3$ per shard, S-BAC eventually decides. This is a direct consequence of BFT consensus liveness per shard [Dwork et al.(1988)] combined with the non-blocking atomic commit structure [Al-Bassam et al.(2018)]. Cross-shard deadlock cannot arise because shards do not hold resources across transactions (section 6.2). This argument depends on three premises: (1) the S-BAC prepare phase has bounded duration (enforced by timeout), (2) the abort constructor unconditionally releases locks (theorem 3.7), and (3) the S-BAC coordination protocol is itself deadlock-free under partial synchrony.*

The deadlock-freedom argument rests on a structural property of the UTxO model: transactions consume *inputs and* create *outputs in a single atomic step, so no shard ever holds a resource while waiting for another shard to release one. This eliminates the hold-and-wait condition that is necessary for deadlock in Coffman's classical characterisation [Coffman et al.(1971)]. By contrast, account-model systems (e.g., Ethereum 2.0 with cross-shard calls) must contend with hold-and-wait whenever a contract on shard A locks state and issues a synchronous call to shard B. In ICE-UTxO, the coordination witness (section 6.2) encodes all cross-shard dependencies as a static DAG before any shard begins execution, so the S-BAC prepare phase can proceed in parallel without circular waits.*

4.6.5 Model Checking Validation

TLC model checking found no counterexamples in state spaces up to MAX_UTXOS=3, MAX_PENDING_TX
This provides counterexample-search confidence, not proof-level confidence; full mechanization of temporal liveness remains future work. The TLA+ specification defines `FairSpec` with weak fairness on commit, abort, and rollback actions, and `StrongFairSpec` with additional strong fairness on `HandleTxEffect`. TLC confirms that `LIVE_TxEventuallyTerminates`, `LIVE_EffectsEventuallyHandled`, and `LIVE_CanReturnToIdle` hold under these specifications. The TLA+ specification comprises approximately 4,600 lines across 18 modules, with 47 named safety invariants and 3 liveness properties.

5 Conflict Serializability

In a sharded ledger, multiple validators or shards may apply transactions in different orders. Conflict serializability guarantees that all such orderings produce the same final state, provided they respect the causal dependencies between transactions. Without this guarantee, shards that process non-conflicting transactions in parallel could diverge on asset ownership — a fatal property for any distributed ledger.

Consider two shards that both observe transactions t_1 and t_2, where t_1 and t_2 do not conflict. Shard A applies them in order $[t_1, t_2]$; Shard B in order $[t_2, t_1]$. If the resulting ledger states differ, the system has forked—a fatal inconsistency. Conflict serializability prevents this: non-conflicting transactions must commute, guaranteeing that all shards arrive at identical state regardless of ordering.

Classical conflict serializability, due to Papadimitriou [Papadimitriou(1986)] and Bernstein et al. [Bernstein et al.(1987)], establishes that a history is serializable if its conflict graph is acyclic: some serial ordering of the transactions produces the same outcome as the concurrent execution. The result in this section is stronger. We prove not merely that *some* serial order exists, but that *all* conflict-respecting orders produce *identical* core state. This universal quantification is what makes the result useful for sharded execution: it is not enough to know that a good ordering exists; every shard must arrive at the same state regardless of which valid ordering it happens to use.

The proof relies on a deterministic-transaction assumption: in the ICE-UTxO model, a transaction's state transition depends only on its declared inputs, outputs, and readSet — not on block position, timestamp, or other hidden state. This is what makes the conflict relation complete: if two transactions can produce different outcomes depending on their relative order, they must share a resource (an input, output, or read/write set element), and that sharing is captured by *fullConflicts* (theorem 5.2). The proof technique is the classical bubble-sort / adjacent-swap method from Mazurkiewicz trace theory [Mazurkiewicz(1987)]: two sequences are equivalent if they differ only by transpositions of adjacent independent (non-conflicting) actions. The contribution here is mechanizing this argument end-to-end in Lean for a full UTxO model with proof-carrying semantics, where all results are verified in `StarstreamPilot.lean`.

5.1 Precedence Graph and Conflict Relations

Definition 5.1 (Conflicts)**.** *Two transactions conflict if they have overlapping read/write sets:*

$$\mathit{conflicts}(t_1, t_2) \iff (t_1.\mathit{writeSet} \cap t_2.\mathit{readSet}) \neq \emptyset \vee (t_1.\mathit{readSet} \cap t_2.\mathit{writeSet}) \neq \emptyset \vee (t_1.\mathit{writeSet} \cap t_2.\mathit{writeSet}) \neq$$

Definition 5.2 (Full Conflicts)**.**

$$\begin{aligned} \mathit{fullConflicts}(t_1, t_2) \iff\ & \mathit{conflicts}(t_1, t_2) \vee \mathit{conflicts}(t_2, t_1) \\ & \vee (t_1.\mathit{outputs} \cap t_2.\mathit{inputs}) \neq \emptyset \\ & \vee (t_2.\mathit{outputs} \cap t_1.\mathit{inputs}) \neq \emptyset \\ & \vee (t_1.\mathit{outputs} \cap t_2.\mathit{outputs}) \neq \emptyset \end{aligned}$$

The *fullConflicts* relation captures every way two transactions can interfere: read-write overlap (one reads what the other writes), write-write overlap (both modify the same state element), input-output overlap (one consumes what the other produces), and output-output overlap (both claim the same fresh identifier). The output-output clause covers the case where two transactions produce the same output identifier — unlikely when identifiers are hash-derived, but included for completeness to ensure the conflict relation is sound in all cases. The relation is deliberately broad: any pair of transactions *not* flagged by *fullConflicts* can be applied in either order with identical core-state results (theorem 5.8). If the conflict relation were incomplete —

missing some form of interference — then the serializability proof would be unsound, because a "non-conflicting" swap could change the outcome.

Theorem 5.3 (Symmetry)**.** $\mathit{fullConflicts}(t_1, t_2) \implies \mathit{fullConflicts}(t_2, t_1)$. (Mechanized: `fullConflicts_symm`

Definition 5.4 (Precedence)**.** *For a history hist, define* $\mathit{before}(\mathit{hist}, t_1, t_2) \iff \mathit{idxOf}(t_1) < \mathit{idxOf}(t_2)$ *and:*

$$\mathit{fullPrecEdge}(\mathit{hist}, t_1, t_2) \iff \mathit{before}(\mathit{hist}, t_1, t_2) \wedge \mathit{fullConflicts}(t_1, t_2)$$

Definition 5.5 (Acyclicity)**.** $\mathit{fullPrecGraphAcyclic}(\mathit{hist}) \iff \forall t.\, \neg\, \mathit{TransGen}(\mathit{fullPrecEdge}(\mathit{hist}))(t, t)$.

Together, *fullPrecEdge* constructs a directed graph over the committed history where an edge from t_1 to t_2 means t_1 was committed before t_2 and they conflict. Acyclicity of this precedence graph is the key safety condition: it is enforced by the commit rule (section 4) as part of the ledger invariant. A cycle would mean two transactions each "depend" on the other having executed first — an impossible situation that would make the history inherently non-serializable. The acyclicity check is not an additional proof obligation but a consequence of the well-formedness conditions already established in section 4.

5.2 Core State Abstraction and Commutativity

The key insight enabling serializability proofs is to project away the history from the ledger, yielding a *core state* where non-conflicting commits can be shown to commute. The full ledger state includes the history list (whose ordering is definitional and thus inherently order-dependent), locked sets (ephemeral concurrency-control state), and pending transactions (irrelevant to committed outcomes). The core state retains only the UTxO set and consumed set—the minimal information needed to determine asset ownership. On this projection, the commutativity proof becomes tractable.

Definition 5.6 (Core State)**.** $\mathit{CoreState} = (\mathit{utxos} : \mathcal{P}_{\mathit{fin}}(\mathit{UTXOId}),\ \mathit{consumed} : \mathcal{P}_{\mathit{fin}}(\mathit{UTXOId}))$. *The projection* $\mathit{coreOf}(L) = (L.\mathit{utxos}, L.\mathit{consumed})$ *extracts core state from a ledger.*

Definition 5.7 (Core Commit)**.** $\mathit{applyCoreCommit}(s, tx) = ((s.\mathit{utxos} \setminus tx.\mathit{inputs}) \cup tx.\mathit{outputs},\ s.\mathit{consumed} \cup tx.\mathit{inputs})$.

Theorem 5.8 (Core Commutativity)**.** *Non-conflicting transactions commute at the core state level:*

$$\neg\, \mathit{fullConflicts}(t_1, t_2) \implies \mathit{applyCoreCommit}(\mathit{applyCoreCommit}(s, t_1), t_2) = \mathit{applyCoreCommit}(\mathit{applyCoreCom$$

(Mechanized: `core_commute`, StarstreamPilot.lean.)

Proof sketch. From the five negated disjuncts of $\neg\, \mathit{fullConflicts}$, we extract $t_1.\mathit{outputs} \cap t_2.\mathit{inputs} = \emptyset$ and $t_2.\mathit{outputs} \cap t_1.\mathit{inputs} = \emptyset$. The proof proceeds by extensional reasoning on `Finset` membership for both components. For `utxos`: $((A \setminus B_1) \cup C_1) \setminus B_2) \cup C_2 = ((A \setminus B_2) \cup C_2) \setminus B_1) \cup C_1$ when $C_1 \cap B_2 = \emptyset$ and $C_2 \cap B_1 = \emptyset$. For `consumed`: union is commutative and associative. □

The commutativity result depends critically on the core state abstraction. The core state strips away the history list, pending transactions, and lock state — retaining only the UTxO set and consumed set. This is what makes the commutativity proof tractable: at the core level, a commit is a pair of set operations (remove inputs from UTxOs, add outputs to UTxOs, union inputs into consumed), and set operations on disjoint elements commute. The full ledger state includes the history list (which is order-dependent by construction) and lock state (which is ephemeral), neither of which affects the final asset ownership that serializability must protect.

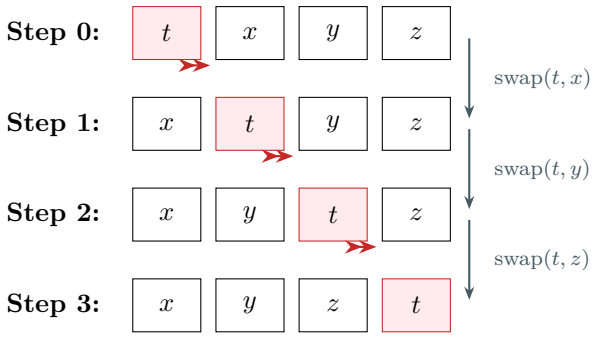

Figure 8: Bubble-sort swap illustration. Transaction t (highlighted) does not conflict with x, y, or z, so iterated adjacent transpositions move t to the end.

Theorem 5.9 (Adjacent Swap)**.** *Swapping adjacent non-conflicting transactions preserves core history:*

$$\neg\, \mathit{fullConflicts}(t_1, t_2) \implies \mathit{applyCoreHistory}(s, \mathit{pre} +\!\!+ [t_1, t_2] +\!\!+ \mathit{suf}) = \mathit{applyCoreHistory}(s, \mathit{pre} +\!\!+ [t_2, t_1] +\!\!+ \mathit{suf})$$

(Mechanized: `core_swap_nonconflicting`.)

5.3 Strong Serializability via Bubble-Sort

We now build the machinery to prove that *all* conflict-respecting permutations produce the same core state.

Definition 5.10 (Conflict Equivalence)**.** *The inductive relation ConflictEquiv on transaction lists is generated by reflexivity, adjacent swap of non-conflicting transactions, and transitivity.*

Theorem 5.11 (Conflict Equivalence Preserves Core State)**.** $\mathit{ConflictEquiv}(\ell_1, \ell_2) \implies \mathit{applyCoreHistory}(s, \ell_1$ $\mathit{applyCoreHistory}(s, \ell_2)$. (Mechanized: `conflict_equiv_same_core`.)

Lemma 5.12 (Bubble Past Suffix)**.** *If t does not fully conflict with any element of suf, then* $\mathit{ConflictEquiv}(\mathit{pre} +\!\!+ [t] +\!\!+ \mathit{suf},\ \mathit{pre} +\!\!+ \mathit{suf} +\!\!+ [t])$. (Mechanized: `bubble_past_suffix`.)

Definition 5.13 (Respects Conflict Order)**.** $\mathit{respectsConflictOrder}(\mathit{order}, \mathit{hist}) \iff \forall t_1, t_2.\ \mathit{fullConflicts}(t_1, t_2)$ $\mathit{before}(\mathit{hist}, t_1, t_2) \implies \mathit{before}(\mathit{order}, t_1, t_2)$

Definition 5.14 (Strong Core Serializability)**.** $\mathit{strongCoreSerializable}(s_0, \mathit{hist}) \iff \forall \mathit{order}.\ \mathit{Perm}(\mathit{order}, \mathit{hist})$ $\mathit{respectsConflictOrder}(\mathit{order}, \mathit{hist}) \implies \mathit{applyCoreHistory}(s_0, \mathit{order}) = \mathit{applyCoreHistory}(s_0, \mathit{hist})$

The definitions of *respectsConflictOrder* and *strongCoreSerializable* quantify over permutations of the history, which raises a natural question: are conflict-respecting permutations actually *executable*? That is, if we applied the transactions in such a permutation, would each transaction find its required inputs available? The answer is yes, by construction of the conflict relation. If transaction B consumes an output produced by transaction A, then $A.\mathit{outputs} \cap B.\mathit{inputs} \neq \emptyset$, so $\mathit{fullConflicts}(A, B)$ holds. Since A was committed before B in the original history, *respectsConflictOrder* forces A before B in every considered permutation. More generally, every causal dependency between transactions is captured by a conflict edge, so no conflict-respecting permutation can place a transaction before the transaction it depends on.

Theorem 5.15 (Acyclic Implies Strong Serializability)**.**

$$\mathit{hist}.\mathit{Nodup} \wedge \mathit{fullPrecGraphAcyclic}(\mathit{hist}) \implies \mathit{strongCoreSerializable}(s_0, \mathit{hist})$$

(Mechanized: `acyclic_strong_serializable`, StarstreamPilot.lean.)

Proof. By strong induction on $|hist|$ using `List.reverseRecOn` (which decomposes the list by peeling the last element, aligning the Lean proof structure with the paper's "bubble to end" argument).

Base case ($hist = []$): any permutation of [] is []; trivial.

Inductive case ($hist = init +\!\!+ [t]$): given a conflict-respecting permutation *order* of $init +\!\!+ [t]$:

1. **Locate** t: since $t \in order$, write $order = pre +\!\!+ [t] +\!\!+ suf$.
2. **Suffix is non-conflicting**: for each $x \in suf$, we have $before(order, t, x)$. If $fullConflicts(t, x)$ held, then by symmetry $fullConflicts(x, t)$, and since $x \in init$ we would have $before(hist, x, t)$, so *respectsConflictOrder* would force $before(order, x, t)$—contradicting $before(order, t, x)$.
3. **Bubble** t **to end**: by theorem 5.12, $ConflictEquiv(order, pre +\!\!+ suf +\!\!+ [t])$.
4. **Core state equivalence**: by theorem 5.11, $applyCoreHistory(s_0, order) = applyCoreHistory(s_0, pre +\!\!+ suf +\!\!+ [t])$.
5. **Apply IH**: $pre +\!\!+ suf$ is a permutation of *init* that respects its conflict order. By the induction hypothesis, $applyCoreHistory(s_0, pre +\!\!+ suf) = applyCoreHistory(s_0, init)$.
6. **Chain**: $applyCoreHistory(s_0, order) = applyCoreHistory(s_0, init +\!\!+ [t]) = applyCoreHistory(s_0, hist)$.

□

Corollary 5.16 (Strong Serializability from Ledger Invariant)**.** $ledgerInvariant(L) \implies strongCoreSerializable$ (Mechanized: `acyclic_precgraph_strong_serializable`.)

The universal diamond property: the committed history's core state is an invariant of the conflict-order class. Any two conflict-respecting serializations produce identical UTxO and consumed sets. This result is precisely the confluence property from Mazurkiewicz trace theory [Mazurkiewicz(1987)]: the relation *ConflictEquiv* (theorem 5.10) is trace equivalence induced by the independence relation $\neg\, fullConflicts$, and theorem 5.15 states that all traces in the same equivalence class produce the same core state. The "diamond" in "universal diamond property" is the same diamond as in Newman's lemma from abstract rewriting: if a single adjacent swap preserves the result (the local confluence step), then any finite sequence of swaps does too (global confluence). The mechanized proof makes this reasoning explicit through the inductive structure of *ConflictEquiv*.

The scope of the invariance is the *core state*: the UTxO set and consumed set, which together determine asset ownership. This is the security-critical state. History ordering and auxiliary ledger fields (e.g., the ordered list of committed transactions, pending locks) may differ across conflict-respecting serializations, but the state that determines who owns what is invariant. For cross-shard consistency, this is the property that matters: different shards can apply non-conflicting transactions in different orders and still agree on asset ownership, without requiring a single global serialization.

Deployed systems rely on the same commutativity principle, albeit without formal proof. Solana's Sealevel runtime [Yakovenko(2018)] schedules non-conflicting account transactions in parallel across cores. Sui [Blackshear et al.(2023)] bypasses consensus entirely for owned-object transactions, exploiting the fact that non-conflicting operations commute. Aptos Block-STM [Gelashvili et al.(2023)] uses optimistic parallel execution with abort-on-conflict, re-executing only when a conflict is detected. All three designs implicitly assume what Core Commutativity (theorem 5.8) formalizes. The mechanized proof in this section provides the missing formal justification for the commutativity assumption that underlies these systems.

6 MPST-to-Ledger Bridge

The preceding sections developed two formal layers independently. The MPST coordination layer (section 3) defines global scripts, event structures, role projections, and local conformance—the machinery for specifying and checking multiparty protocols. The ledger commit layer (section 4) defines the operational semantics, commit guards, invariant preservation, and the conflict serializability results of section 5. The bridge problem is to connect these layers: to show that a transaction that commits on the ledger actually followed the prescribed multiparty protocol (soundness), and that any protocol-compliant execution can lead to a valid commit (completeness). Without this connection, the two layers would be independent formalisms—the MPST types would guarantee protocol structure but say nothing about ledger state, and the ledger invariants would guarantee asset safety but say nothing about whether the protocol was followed.

The bridge has three parts, corresponding to the three subsections. First, *trace consistency* (section 6.1) identifies conditions under which shard-local verification—checking each role's projected trace independently—can reconstruct the global trace consistency needed for commit. This reduces a global property (the entire multiparty trace is valid) to a conjunction of local properties (each role's projection conforms) plus lightweight cross-role conditions. Second, *coordination witnesses* (section 6.2) connect these trace-level results to the ledger's commit mechanism: the bidirectional witness theorems show that a coordination witness is globally valid if and only if it passes all shard-local checks, and the coordinated commit theorem shows that the global commit guard implies all local commit guards. Third, *concurrent-to-serial refinement* (section 6.3) shows that the concurrent ledger execution—with its interleavings of effect handling, lock acquisition, and coroutine scheduling—refines a serial specification where transactions commit one at a time.

The refinement proof follows the classical stuttering simulation approach: an abstraction map projects the concrete ledger state onto a serial specification state, concrete steps that only modify concurrency-control metadata are classified as stuttering (invisible to the specification), and each concrete commit step is shown to correspond to exactly one serial step. This technique is standard in distributed systems verification—Lamport's TLA [Lamport(1994)] formalizes stuttering refinement for state machines, and IronFleet [Hawblitzel et al.(2015)] and Verdi [Wilcox et al.(2015)] use similar simulation proofs to connect protocol specifications to implementation code. The novelty here is combining stuttering simulation with MPST-based protocol verification: the refinement witness for each commit step includes not just ledger-level validity (inputs live, outputs fresh, no double-spend) but also a cryptographic attestation that the multiparty protocol was correctly executed.

6.1 Trace Consistency and Cross-Role Reconstruction

A central question is: can shard-local verification (checking each role's local trace) reconstruct the global trace consistency needed for commit?

Definition 6.1 (Before)**.** $\mathit{Before}(tr, a, b) \iff \exists \ell_1, \ell_2.\ tr = \ell_1 +\!\!+ [a] +\!\!+ \ell_2 \wedge b \in \ell_2$.

Definition 6.2 (Trace Consistency)**.** *A trace tr is* consistent *with script S, written $S.\mathit{traceConsistent}(tr)$, if:*

1. *tr is duplicate-free ($tr.\mathit{Nodup}$)*
2. *All events in tr are in $S.\mathit{events}$*
3. *No event in tr conflicts with any other event in tr*
4. *For every order edge $e' < e$ with $e \in tr$, $\mathit{Before}(tr, e', e)$*

These four conditions are the standard requirements for a *configuration* of an event structure [Winskel(1987)]: no duplicates (each event occurs at most once), membership in the event set (only events from the script), conflict-freedom (no two mutually exclusive events co-occur), and causal closure (if an event is included, all its causal predecessors are included in the correct order). A trace satisfying these conditions is a linearization of such a configuration—it represents an executable prefix of the protocol in which every event finds its prerequisites already completed. The distinction between *traceConsistent* and *validTrace* (theorem 6.5) is that the former is a static structural check on a list of events, while the latter asserts that the trace can be produced by the operational semantics of the event structure. Theorem 6.5 proves these coincide: the structural conditions are exactly the enablement conditions, so checking a trace against theorem 6.2 is equivalent to replaying it step by step against the event-structure semantics.

Definition 6.3 (Cross-Role Consistency)**.** *A trace is* cross-role consistent *if:*

1. *Events with disjoint role sets do not conflict:* $\mathit{disjointRoles}(e, f) \implies \neg(e \# f)$.
2. *Order between disjoint-role events is witnessed:* $\mathit{disjointRoles}(e', e) \wedge e' < e \wedge e \in tr \implies \mathit{Before}(tr, e', e)$.

Local conformance per role (*localConform*) ensures that each participant follows its own projected protocol—events involving role r appear in the correct order and without intra-role conflicts. But local conformance says nothing about the relationship between events on disjoint roles: if event e involves only role r_1 and event f involves only role r_2, no single role's projection sees both. Cross-role consistency (theorem 6.3) fills this gap. Condition (1) requires that events on independent roles do not conflict—a property guaranteed when roles access disjoint UTxO sets (as established by the role-separation results in section 3.6), but stated here as an explicit proof obligation. Condition (2) ensures that any causal ordering the global protocol imposes between events on different roles is respected in the trace, even though no single role observes both events. Together with the four conditions of theorem 6.2, these requirements exactly characterize the linearizations of conflict-free, causally-closed configurations in the event structure induced by the global script, in the sense of Winskel's event-structure semantics [Winskel(1987)]. The case split in the proof of theorem 6.4—shared role vs. disjoint roles—mirrors this decomposition: intra-role properties come from local conformance, and inter-role properties come from cross-role consistency.

Theorem 6.4 (Local + Cross-Role ⇒ Global Consistency)**.**

$$\begin{aligned} &\mathit{WF}(S) \wedge tr.\mathit{Nodup} \wedge (\forall r \in S.\mathit{roles}.\ \mathit{localConform}(S, r, \mathit{traceProj}(S, r, tr))) \\ &\quad \wedge \mathit{crossRoleConsistent}(S, tr) \implies S.\mathit{traceConsistent}(tr) \end{aligned}$$

(Mechanized: `traceConsistent_of_local_and_cross`, Script.lean.)

Proof sketch. For *conflict-freedom*: given events a, b in tr, either they share a role or have disjoint roles. If disjoint, cross-role consistency directly gives $\neg(a \# b)$. If they share a role r, then both appear in $\mathit{traceProj}(S, r, tr)$; local conformance implies the local trace is conflict-free, which lifts to the global level.

For *order-respect*: given $e' < e$ with $e \in tr$, case-split on whether e' and e share a role. If disjoint, cross-role consistency provides $\mathit{Before}(tr, e', e)$. If shared via role r, local trace validity ensures e' appears before e in the projected trace, which lifts to the global trace via `before_of_filter`. □

Theorem 6.5 (Consistent Implies Valid Trace)**.** $S.\mathit{traceConsistent}(tr) \implies S.\mathit{validTrace}(tr)$. (Mechanized: `traceConsistent_implies_validTrace`.)

Proof sketch. By induction on tr, converting consistency evidence into enablement at each step: predecessors present (from order-respect), no conflicts (from pairwise conflict-freedom), event membership (from events check). □

Together, theorems 6.4 and 6.5 establish a two-step reduction. Theorem 6.4 reduces global trace consistency—a property of the entire multiparty execution—to a conjunction of per-role local conformance checks and cross-role consistency conditions. Theorem 6.5 then shows that any globally consistent trace is a valid execution of the event structure, meaning it could have been produced by the MPST operational semantics. The practical consequence is that a shard hosting role r needs to check only that r's projected trace conforms locally; the cross-role conditions are enforced by the coordination witness construction (which embeds the global trace) and the cryptographic proof (which attests to the trace's consistency). This decomposition is what makes shard-local verification feasible: without it, every shard would need to replay the entire multiparty execution to decide whether to commit.

6.2 Coordination Witness and Commit

The S-BAC (Sharded Byzantine Atomic Commit) bridge layer connects the MPST coordination results from the previous subsection to the ledger's commit mechanism. A coordination witness $W = (S, tr)$ pairs a global script with a trace attesting to protocol compliance. The question is whether checking this witness can be decomposed across shards: does global validity of the witness imply that each shard's local check will pass, and conversely does passing all local checks imply global validity? The following two theorems answer both directions affirmatively.

Theorem 6.6 (Bidirectional Witness Theorems)**.** *1.* Top-down*: $witnessGlobalOK(W) \implies witnessLocalOK(W)$. A globally valid witness can be verified shard-locally.*

2. Bottom-up*: $WF(W.script) \wedge witnessConsistent(W) \implies witnessGlobalOK(W)$. If the witness trace is consistent, global conformance follows.*

Theorem 6.7 (Coordinated Commit Implies Local Commit)**.** $coordCommitEnabled(m, L, ctx) \implies coordCommitEnabledLocal(m, L, ctx)$. (Mechanized: `coordCommitEnabledLocal_of_global`.)

Cross-shard deadlock. The S-BAC protocol is deadlock-free by construction: each transaction is submitted to all relevant shards simultaneously during prepare, and shards respond independently. There is no circular wait because shards do not hold resources across transactions—a shard either prepares or aborts, and the final decision is made by the coordinator after collecting all responses. This argument depends on three premises: (1) the S-BAC prepare phase has bounded duration (enforced by timeout), (2) the abort constructor unconditionally releases locks (theorem 3.7), and (3) the S-BAC coordination protocol is itself deadlock-free under partial synchrony.

The bidirectional witness theorems (theorem 6.6) establish that global protocol validity and shard-local validity are equivalent, given well-formedness and trace consistency. The top-down direction guarantees that a globally valid witness will pass every shard's local checks—no honest shard will reject a legitimate transaction. The bottom-up direction guarantees the converse: if every shard independently approves its roles' projections and the trace is consistent, then the global protocol was followed. Together, these two directions justify the decentralized verification architecture: no single authority needs to check the global protocol end-to-end. Each shard verifies only the roles it hosts, and the trace consistency conditions (theorems 6.2 and 6.3) ensure that these local checks compose into a global guarantee. This decomposition is critical for scalability—global protocol checking would require every shard to see every event, defeating the purpose of sharding.

Theorem 6.7 extends this result to the commit layer: if the full coordinated commit guard passes (combining both ledger-level validity and witness-level protocol compliance), then every shard's local commit condition is satisfied. This means the S-BAC prepare phase will succeed on all honest shards—no shard will unilaterally abort a transaction that the global commit guard has approved. The converse direction (all-local-approve implies global-approve) follows from the bottom-up witness theorem combined with the structure of *coordCommitEnabled*: if every shard's local ledger checks and local witness checks pass, and the trace is consistent, then both *commitEnabledStrong* and *witnessGlobalOK* hold, satisfying the global guard.

6.3 Concurrent-to-Serial Refinement

We prove that the concurrent ICE-UTxO semantics refines a serial specification where transactions commit atomically one at a time.

Definition 6.8 (Serial Step)**.** *A serial step atomically commits a proof-verified, valid transaction:*

$$\mathit{SerialStep}(m, L, L') \iff \exists tx.\ \mathit{allProofsVerified}(tx) \wedge \mathit{validTx}(L, tx) \wedge tx.\mathit{phase} = \mathit{Committing} \wedge L' = \mathit{applyComm}$$

Definition 6.9 (Stuttering)**.** *A step is* stuttering *if* $\mathit{absLedger}(L) = \mathit{absLedger}(L')$*. Stuttering steps change only the concurrency-control fields without modifying the core UTxO/consumed/history state.*

Definition 6.10 (Abstraction Map)**.** $\mathit{absLedger}(L) = L[\mathit{locked} := \emptyset, \mathit{pending} := \emptyset, \mathit{effects} := \lambda\_.\,[], \mathit{handlerStacks} := \lambda\_.\,[]]$.

The abstraction map erases exactly the concurrency-control fields that exist only in the concurrent implementation: *locked* tracks which UTxOs are reserved during the S-BAC prepare phase (preventing double-spending during coordination), *pending* tracks transactions that have been submitted but not yet committed or aborted, and *effects* and *handlerStacks* manage the algebraic-effect coroutine machinery that allows transactions to yield, raise effects, and resume. None of these fields exist in the serial specification, where each transaction commits instantaneously in a single atomic step. The result is that $\mathit{absLedger}(L)$ retains only the UTxO set, the consumed set, and the committed history—the state components that determine asset ownership and transaction ordering, and the only components that the serial specification tracks.

Theorem 6.11 (Concurrent Refines Serial)**.**

$$\mathit{Steps}(m, L_0, L_n) \wedge \mathit{ledgerInvariant}(L_0) \implies \mathit{SerialSteps}(m, \mathit{absLedger}(L_0), \mathit{absLedger}(L_n))$$

(Mechanized: `concurrent_refines_serial`, StarstreamPilot.lean.)

Proof sketch. By induction on the `Steps` derivation. At each step, apply `step_preserves_invariant` to maintain the invariant for the induction hypothesis. Then case-split on the step constructor:

- **Stuttering cases** (7 of 8): `addPending`, `lockInputs`, `abort`, `installH`, `uninstallH`, `raiseE`, `handleE`. Each has a dedicated lemma showing $\mathit{absLedger}(L) = \mathit{absLedger}(L')$. The serial trace is unchanged.
- **Visible case** (1 of 8): `commit`. The commit step produces a `SerialStep` in the abstract trace, constructed from the commit preconditions.

□

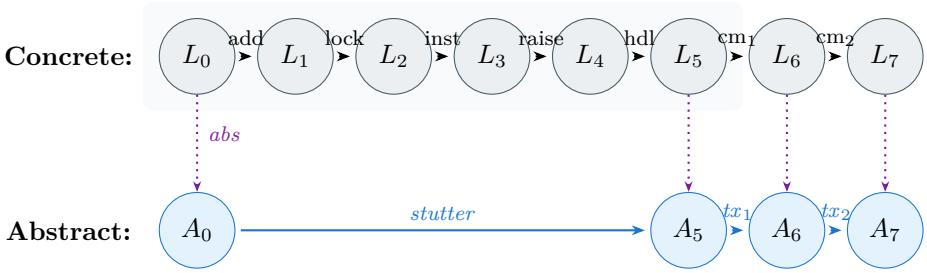

Figure 9: Refinement diagram: concrete concurrent steps map to abstract serial steps via *absLedger*. Non-commit steps are stuttering.

The practical consequence is that the concurrent ledger—with its interleavings of effect handling, lock acquisition, and coroutine scheduling—is an implementation detail. Any concurrent execution simplifies to an equivalent serial execution where transactions commit one at a time. Reasoning about ICE-UTxO's correctness thus reduces to reasoning about sequential commits, which is substantially simpler.

The 7-to-1 split between stuttering and visible steps reflects a design principle: the abstraction map *absLedger* (theorem 6.10) erases exactly the fields that manage concurrency control—locks, pending transactions, effect queues, and handler stacks. The seven non-commit constructors manipulate only these fields; none modify the UTxO set, consumed set, or committed history. Only the *commit* step changes the core state visible to the serial specification.

This decomposition is the standard structure of a stuttering simulation in the sense of Lamport [Lamport(1994)]: the implementation takes many "internal" steps for each "visible" step of the specification. A transaction may raise and handle arbitrarily many effects, install and uninstall multiple handlers, and retry after aborts—an unbounded number of concrete steps—before it eventually commits. The proof does not require a bound on this ratio; it requires only that each concrete step either maps to a serial step (the commit case) or leaves the abstract state unchanged (the stuttering case). The liveness results from section 4.6 ensure that this stuttering does not continue forever: under fair scheduling, every pending transaction eventually commits or aborts, so the concrete execution cannot stutter indefinitely without making abstract progress.

The predicate $\mathit{refinementWitness}(m, L, tx)$ bundles the conditions under which a concrete commit step corresponds to an abstract serial step. Specifically, it requires: (a) the transaction's coordination witness is globally valid ($\mathit{witnessGlobalOK}$), certifying that the MPST protocol was followed; (b) all cryptographic proof commitments have been verified ($\mathit{allProofsVerified}(tx)$); (c) the transaction is structurally valid in the current ledger state ($\mathit{validTx}(L, tx)$)—inputs are live, outputs are fresh, and inputs and outputs are disjoint; and (d) the transaction is in the Committing phase. This is essentially *coordCommitEnabled* (section 3.6) restricted to a single transaction in a given ledger state. The name "circuit witness" reflects the deployment architecture: in practice, MPST trace compliance is attested by an IVC/SNARK proof (the "circuit"), so verifying the *refinementWitness* amounts to checking one cryptographic proof rather than replaying the entire multiparty protocol execution.

Theorem 6.12 (Circuit Witness Implies Serial Step)**.** $\mathit{refinementWitness}(m, L, tx) \implies \mathit{SerialStep}(m, L, \mathit{apply}$
(Mechanized: `circuit_witness_implies_serial_step`.)

The results of this section establish a layered correctness argument for the ICE-UTxO system. Trace consistency (theorems 6.2 and 6.3) and the bidirectional witness theorems (theorem 6.6) ensure that shard-local verification is both sound and complete with respect to the global MPST protocol: a transaction commits only if its coordination witness certifies a protocol-compliant execution, and every protocol-compliant execution produces a witness that passes shard-local checks. The concurrent-to-serial refinement (theorem 6.11) then shows that

the committed transactions, viewed through the abstraction map, form a serial history—each transaction appears to execute atomically and in isolation. Combined with the conflict serializability result from section 5, this gives a two-level guarantee: the committed history is serializable (section 5), and each transaction in that history faithfully implements its declared multiparty protocol (this section).

The two-layer architecture mirrors the approach of verified distributed systems such as IronFleet [Hawblitzel et al.(2015)] and Verdi [Wilcox et al.(2015)]. In IronFleet, a high-level protocol specification (e.g., a Paxos state machine) is connected to a low-level implementation via a simulation proof; in Verdi, verified system transformers lift correctness from idealized models to realistic network settings. Here, the MPST event-structure semantics serves as the protocol specification layer, and the concrete UTxO operational semantics serves as the implementation layer. Theorem 6.11 is the simulation theorem connecting them. The key difference is that ICE-UTxO uses cryptographic proof-carrying data to certify protocol compliance at commit time, rather than relying on runtime message monitoring. This shifts the trust boundary: instead of trusting that each participant runs correct code (as in IronFleet), the ledger verifies a cryptographic attestation that the protocol was followed, regardless of how participants behaved internally.

The bridge relies on assumptions stated in section 4.2: (i) ZK verifier soundness—the external ZK verification oracle is assumed to be sound, so that a `Verified` flag genuinely corresponds to a valid proof of protocol compliance; (ii) S-BAC shard honesty—the cross-shard atomic commit protocol assumes fewer than one-third Byzantine validators per shard, consistent with the Chainspace fault model [Al-Bassam et al.(2018)]; and (iii) well-formedness of the global script ($WF(S)$), which is a checkable structural condition on the MPST global type. The S-BAC protocol itself is not mechanized in Lean; its correctness is assumed from the established literature on Byzantine atomic commit [Al-Bassam et al.(2018)]. Formally verifying the S-BAC layer—perhaps using a framework like Verdi's verified system transformers—would close this remaining gap between the mechanized proofs and a deployed system.

7 Lean Mechanization

The Lean 4 mechanization proves three main results about the ICE-UTxO model: (1) a six-part safety invariant is preserved by every state transition; (2) committed transaction histories are conflict-serializable — every conflict-respecting reordering produces the same asset state; and (3) the concurrent execution, with its interleavings of lock acquisition, effect handling, and coroutine scheduling, refines a serial specification where transactions commit one at a time. A bridge layer connects these ledger-level results to the MPST coordination layer, proving that protocol compliance decomposes into shard-local checks. Every proof is fully discharged — no `sorry`, `admit`, or `axiom` — and the only classical reasoning is a single decidable case split on finite role sets.

The development follows a two-track architecture. Track 1 (Ledger) reasons about UTxO state, conflicts, and serializability; Track 2 (Coordination) reasons about event structures, projections, and protocol compliance. The two tracks share a narrow interface — the *coordCommitEnabled* predicate — which conjoins both tracks' preconditions into a single commit guard. This separation reflects the independence of asset-safety reasoning (which cares about UTxO sets and conflicts) from protocol-compliance reasoning (which cares about event orderings and role projections). Changes to one track do not require re-proving results in the other.

A parallel TLA+ specification complements the Lean development. Lean proves safety properties universally (for all executions); TLA+ validates liveness properties (eventual commit, eventual termination, eventual effect handling) by model checking on bounded instances under fairness assumptions. The two artifacts cross-reference: each TLA+ liveness property names the Lean lemmas it depends on, and the Lean development proves those lemmas without assuming

fairness.

7.1 Proof Architecture

The mechanization is split into two tracks connected by a bridge. This is not a code-organization convenience — it reflects a fundamental separation between two proof domains that share a narrow interface.

Track 1 (Ledger). Track 1 reasons about UTxO sets, conflicts, invariants, and serializability. Its central objects are: `Step` (an inductive with eight constructors, one per state transition: `addPending`, `lockInputs`, `commit`, `abort`, `installH`, `uninstallH`, `raiseE`, `handleE`); *ledgerInvariant* (a six-part safety predicate preserved by every step); *CoreState* (the abstraction that makes commutativity provable); and *absLedger* (the refinement map that drops concurrency-control fields). The key design insight is a double abstraction: *CoreState* drops history so that applying transactions in different orders *can* produce equal states (enabling the serializability proof), while *absLedger* drops pending, locked, effects, and handler stacks so that non-commit steps *do* produce equal states (enabling the stuttering simulation).

Track 2 (Coordination). Track 2 reasons about event structures, projections, traces, and protocol compliance. Its central objects are: `Script` (an event structure with partial order and conflict relation); `LocalScript` (per-role projection); `Program` (PTB commands with four dependency relations — data, ordering, UTxO, and interface — that combine into *orderRel* and *conflictRel*); and `CoordWitness` (a script–trace pair attesting to protocol compliance). The key insight is that global trace consistency decomposes into per-role local conformance plus cross-role consistency, which is what makes shard-local verification possible without replaying the entire multiparty execution.

The bridge. The two tracks meet at the predicate *coordCommitEnabled*, which conjoins *commitEnabledStrong* (the ledger-track guard: proof verified, inputs live, outputs fresh, phase is committing, not in history) and *witnessGlobalOK* (the coordination-track guard: the trace globally conforms to the script). This is the sole point where both proof domains interact. The bridge theorems then show this conjunction decomposes to shard-local checks: each shard verifies only the roles it hosts and the UTxOs it manages, and these independent checks compose into the global guarantee.

The Oracle subtrack. An independent executable model (`Starstream/Oracle/*.lean`) maintains its own type universe with `Bool`-valued predicates and gas metering. Its four-phase proof decomposition (Phases A–D) verifies transaction validation from input checking through batch processing. The PhaseD bridge theorems lift key Starstream properties (`commit_preserves_no_double_spend`, `step_preserves_invariant`, `concurrent_refines_serial`, among others) to the Oracle namespace, each requiring only a single `simpa using` tactic — confirming that the executable model adds no new proof obligations to the core theory.

7.2 What Is Proved

The mechanization establishes three main results and connects them through a bridge layer. The three results are: (1) the concurrent ledger preserves a six-part safety invariant across every state transition; (2) committed transaction histories are conflict-serializable — every conflict-respecting reordering produces the same asset state; and (3) the concurrent execution refines a serial specification via stuttering simulation. The bridge layer proves that global protocol compliance decomposes into shard-local checks.

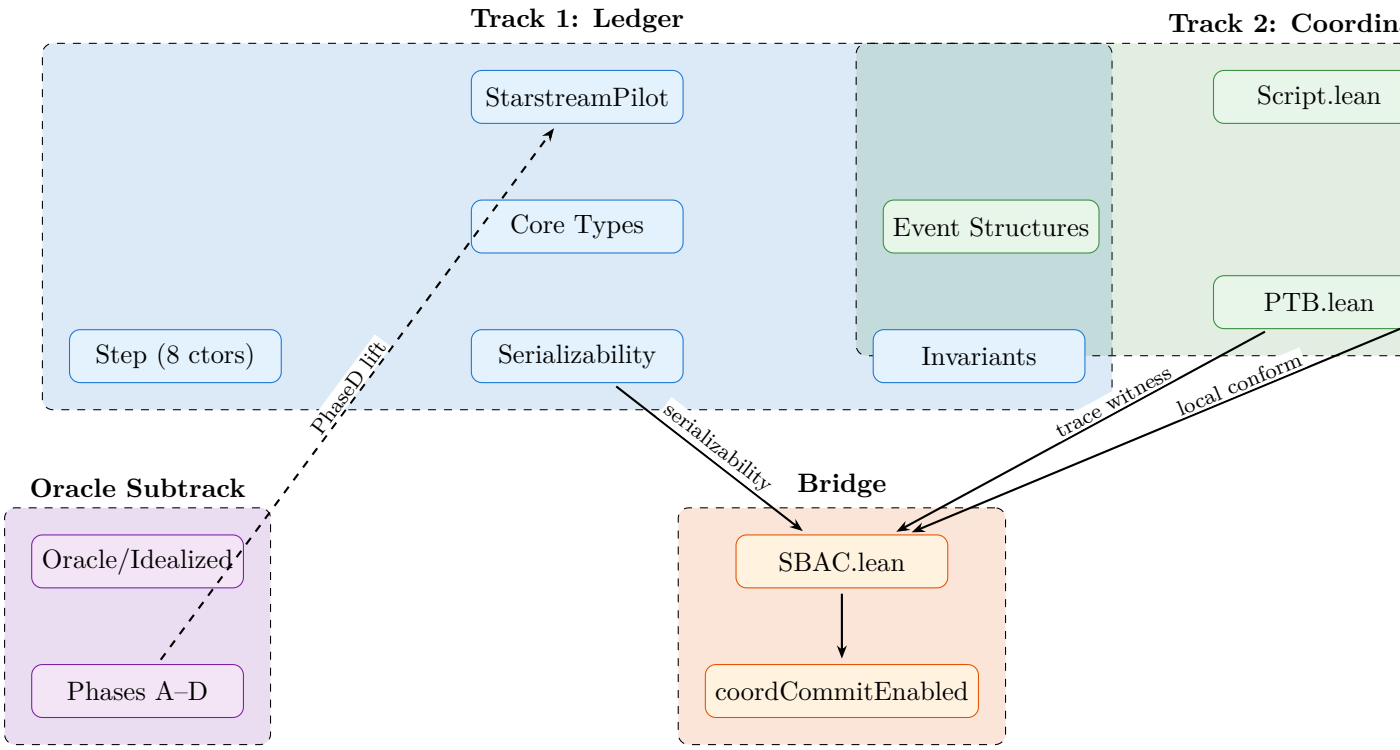

Figure 10: Proof architecture of the Lean mechanization. The two tracks meet at *coordCommitEnabled* through the SBAC bridge. The Oracle subtrack connects back to Track 1 via PhaseD bridge theorems.

The safety invariant. The predicate *ledgerInvariant* is a six-part conjunction: *noDoubleSpend* (UTxOs and consumed are disjoint — no asset duplication); *lockedSubsetActive* (locked UTxOs exist in the active set — no phantom locks); *historyNodup* (committed history has no duplicates — no transaction replay); *committedImpliesVerified* (every transaction in history passed proof verification — proof-gated commit); *precGraphAcyclicExt* (extended precedence graph is acyclic — serializability precondition); and *fullPrecGraphAcyclic* (full-conflict precedence graph is acyclic — strong serializability precondition). The theorem *step_preserves_invariant* proves that every `Step` constructor preserves all six components. *init_ledgerInvariant* establishes the invariant for initial states. *ledgerInvariant_steps* lifts preservation to multi-step executions. Together, these prove that the invariant holds in every reachable state.

Conflict serializability. The theorem *core_commute* proves that non-conflicting transactions produce the same *CoreState* regardless of application order. *CoreState* retains only UTxOs and consumed — *not* history — which is what makes commutativity possible (history is ordered by construction, so swapping two transactions always changes the history). The theorem *acyclic_strong_serializable* then proves the diamond property: if the full-conflict precedence graph is acyclic, then *all* conflict-respecting permutations of the committed history produce the same *CoreState*. This is strictly stronger than existential serializability (there *exists* an equivalent serial order) — it says *every* legal reordering agrees.

Concurrent-to-serial refinement. The theorem *concurrent_refines_serial* proves that for any concurrent execution from an initial state satisfying *ledgerInvariant*, there exists a serial execution producing the same abstract state. The proof classifies each of the eight step types as either stuttering (no change to abstract state) or visible (one serial step). The theorem *circuit_witness_implies_serial_step* connects individual commit steps to the IVC proof

Table 5: Principal theorems in the Lean mechanization.

Theorem	File	Statement (informal)	Te
`core_commute`	StarstreamPilot	Non-conflicting txs commute at CoreState	Fir
`acyclic_strong_serializable`	StarstreamPilot	Acyclic ⇒ all conflict-respecting perms agree	Bu
`commit_requires_proof`	StarstreamPilot	Only proof-verified txs extend history	8-v
`step_preserves_invariant`	StarstreamPilot	Every Step preserves 6-part invariant	Flo
`concurrent_refines_serial`	StarstreamPilot	Concurrent refines serial via stuttering	Stu
`proj_localConform_of_globalConform`	Script	Global conformance ⇒ local conformance	Str
`traceConsistent_of_local_and_cross`	Script	Local + cross-role ⇒ global consistency	De
`toScript_wellFormed`	PTB	PTB programs produce well-formed scripts	7-v
`witnessLocalOK_of_global`	SBAC	Global witness ⇒ local witness	Pro
`witnessGlobalOK_of_local_and_consistent`	SBAC	Consistent local ⇒ global	Co
`coordCommitEnabledLocal_of_global`	SBAC	Global commit ⇒ all-local commit	De

circuit: a valid *refinementWitness* — combining circuit verifiability, structural validity, and mode-specific checks — implies a valid *SerialStep*.

Coordination decomposition. The bidirectional witness theorems prove that checking protocol compliance can be decomposed across shards. *witnessLocalOK_of_global*: if the global witness is valid, every shard's local check passes. *witnessGlobalOK_of_local_and_consistent*: if all local checks pass and the trace is consistent, the global witness is valid. *coordCommitEnabledLocal_of_glob* the global commit guard implies every shard's local commit guard. These justify the S-BAC architecture: each shard checks only its roles, and the local checks compose into a global guarantee.

Proof-gated commit. The theorem *commit_requires_proof* proves that any step extending the history requires $proofOk(tx)$. The proof is an 8-way case analysis on `Step`: the seven non-commit constructors cannot extend history (proved by the helper *list_eq_self_append_singleton*, which shows a list cannot equal itself with an element appended), and the commit constructor requires *commitEnabledStrong*, which includes *allProofsVerified*.

Consumed monotonicity. The theorems *consumed_monotone_step* and *consumed_monotone_steps* prove that the consumed set only grows — once an asset is spent, it stays spent. The theorem *committed_inputs_no_reuse* uses this together with *noDoubleSpend* to show that a new commit's live inputs cannot overlap previously consumed inputs, closing the circle between the invariant and the monotonicity guarantee.

Table 5 provides a compact reference for the principal theorems.

TLA+ specification. A parallel TLA+ specification (∼4,600 lines across 18 modules) complements the Lean development. The TLA+ model defines 47 named safety invariants and 3 liveness properties (*eventual commit* under weak fairness, *eventual termination* of pending transactions, and *eventual effect handling* when a handler is installed), for which TLC model checking found no counterexamples in bounded state spaces under weak and strong fairness conditions. The division of labor is principled: Lean proves safety properties universally (for all executions), while TLA+ validates liveness properties (which hold under fairness assumptions) by model checking on bounded instances. The two artifacts are cross-referenced in section 4.6: each TLA+ liveness property names the Lean lemmas it depends on, and the Lean development proves those lemmas without assuming fairness.

7.3 How the Proofs Work

Conflict serializability by bubble-sort. The proof that acyclicity implies serializability (*acyclic_strong_serializable*) proceeds by strong induction on history length, using Lean's `List.reverseRecOn` eliminator. At each step, the last-committed transaction t is "bubbled" to the end of any conflict-respecting permutation. The key lemma *no_conflict_after_last* shows that all elements appearing after t in the permutation are non-conflicting with t: if any were conflicting, the conflict-respecting ordering would force t to appear before them, contradicting t's position, and closing a cycle that contradicts acyclicity. The lemma *bubble_past_suffix* then constructs a *ConflictEquiv* chain of adjacent swaps moving t to the end. The inductive hypothesis handles the remaining prefix. The algebraic foundation is *core_commute*: non-conflicting transactions produce identical *CoreState* regardless of order, proved by *Finset* extensionality — the resulting UTxO sets and consumed sets are element-wise equal because disjoint inputs and outputs commute under set difference and union. This constructive approach produces the *specific* swap sequence, not merely an existence assertion.

Invariant preservation by 8-way case analysis. The theorem *step_preserves_invariant* cases on the eight `Step` constructors. The `commit` case is the only one that modifies UTxOs, consumed, and history simultaneously; its proof chains *commit_preserves_no_double_spend* (disjointness of new outputs from consumed and disjointness of remaining UTxOs from consumed), *lockedSubset* adjustment (locked inputs are removed, so locked remains a subset of the active set), and history no-dup (freshness from *commitEnabledStrong* ensures the new transaction is not already in history). The seven non-commit cases are simpler: they modify only concurrency-control fields (locked, pending, effects, handlerStacks), so the invariant components involving UTxOs, consumed, and history are trivially preserved. The `handleE` case requires an explicit field-extraction lemma (*handleEffect_fields*) because *handleEffect* returns an `Option Ledger` through pattern matching, and Lean cannot automatically see that the five preserved fields are unchanged without destructuring the result.

Stuttering simulation for refinement. The theorem *concurrent_refines_serial* proceeds by induction on the *Steps* relation (the reflexive-transitive closure of *Step*). The abstraction map *absLedger* zeros out four concurrency-control fields (`locked`, `pending`, `effects`, `handlerStacks`), retaining only `utxos`, `consumed`, and `history`. For each step: seven private stuttering lemmas (one per non-commit constructor) show $absLedger(L) = absLedger(L')$, and the lemma *absLedger_applyCommit* shows $absLedger(applyCommit(L, tx)) = applyCommit(absLedger(L), tx)$, constructing a *SerialStep* witness. The proof threads the ledger invariant through each step via *step_preserves_invariant*, ensuring the inductive hypothesis applies at every intermediate state. The result: for any concurrent execution from an initial state to a final state, there exists a serial execution — a sequence of atomic commits — that produces the same abstract state.

Trace decomposition in the coordination layer. The coordination proofs use a shared-role/disjoint-role case split. The theorem *traceConsistent_of_local_and_cross* takes per-role local conformance and cross-role consistency as hypotheses. If two events share a common role, local conformance handles them (the role's projected trace includes both events, so the local ordering constraint applies); if they are on disjoint roles, cross-role consistency handles them (the explicit inter-role conditions fill the gap that no single role's projection can see). The theorem *traceConsistent_implies_validTrace* then shows that the four structural conditions of *traceConsistent* — no duplicates, membership, conflict-freedom, and causal closure — coincide with step-by-step operational replay against the event-structure semantics, by induction on trace length. The PTB layer adds *toScript_wellFormed*, proving that the deterministic translation from programs to event structures preserves well-formedness, and *witnessGlobalOK_of*, proving that well-formed programs yield globally valid witnesses.

7.4 Lessons Learned

Constructivity and axiom discipline. A single `classical` invocation (Script.lean, line 622) performs a decidable case split on finite role sets. Everything else is constructive. The payoff is concrete: the bubble-sort serializability proof *constructs* the transposition sequence, producing a specific witness of which adjacent swaps transform one conflict-respecting ordering into another. A classical existence proof would give serializability without this witness. The only axioms in scope are those imported by Mathlib (propositional extensionality, quotient soundness, and function extensionality), none of which are invoked directly.

The identity-permutation false start. The initial `coreSerializable` theorem stated that there *exists* a permutation of the history producing the same `CoreState` — and was proved with the identity permutation. Technically correct, vacuously easy, and useless: it says only that applying the history in its original order yields the original state. The corrected `strongCoreSerializable` universally quantifies over *all* conflict-respecting permutations, requiring that they all produce the same `CoreState`. This is the standard diamond property of conflict serializability [Papadimitriou(1986)]. Lean checked both proofs without complaint; only adversarial review of the *statement* caught the error. The lesson is that type-checkers guarantee proof correctness but not statement adequacy.

The two-track decomposition as a proof management strategy. Changes to event-structure semantics (`Script.lean`) never required re-checking ledger proofs (`StarstreamPilot.lean`), and vice versa. The bridge (`SBAC.lean`) works because the two tracks share only the `CoordTx` type and the `coordCommitEnabled` predicate. This narrow interface made iterative development practical: the serializability proof was reworked three times without touching the coordination layer.

The Oracle as a specification testbed. The Oracle subtrack maintains an independent executable model with `Bool`-valued predicates and gas metering. Its four-phase proof decomposition exposed specification bugs that the `Prop`-valued Starstream predicates could not: when an Oracle proof failed, the counterexample was computable, making diagnosis immediate. The PhaseD bridge theorems then confirmed that every Oracle-level property lifts to the protocol layer, requiring only a single `simpa using` tactic per theorem — evidence that the Oracle model adds no new proof obligations to the core theory.

Complementary mechanizations. Tirore et al. [Tirore et al.(2025)] provide the first mechanized proof of subject reduction for MPST in Coq; their work is complementary to ours and could serve as a certified front-end for session-type checking feeding into ICE-UTxO's ledger-level back-end.

7.5 Trust Boundaries

The mechanization makes three trust-boundary assumptions explicit, marked with inline comments in the source code (labels F3, F5, F9). Each assumption defines the boundary between what the model proves and what it assumes from the external environment.

F3: The verifier oracle. The predicate *allProofsVerified*(tx) checks that every `ProofCommitment` attached to a transaction has reached the `Verified` phase. The model does not define what a valid proof *is* — it trusts that the external verifier labels correctly. Every theorem involving committed transactions (`step_preserves_invariant`, `concurrent_refines_serial`, `commit_requires_proof`) is conditional on this oracle. The practical consequence: if the ZK

circuit has a soundness bug, the ledger's formal guarantees do not hold. The model proves the *if-then* structure; the *if* is a cryptographic assumption.

F5: UTxO immutability justifies read-set validation. The predicate *readSetValid* checks $tx.readSet \subseteq l.utxos$ — a pure existence check. In an account-based model this would be insufficient for snapshot isolation because balances can change between read and commit. In the UTxO model, however, each UTxO is immutable from creation to consumption: its value, datum, and validator reference never change. An existence check therefore *is* a snapshot-consistency check. The function *applyCommit* enforces this by moving inputs atomically from *utxos* to *consumed*, never modifying a UTxO in place.

F9: Asynchronous handler registration. The function *handleEffect* returns *none* when no handler is installed for an interface, preventing the `handleE` step from firing. Effects remain queued until *installH* pushes a handler. The event-structure causal order guarantees that *installH* precedes *handleE* for the same interface; the liveness argument (TLA+ property L3) guarantees eventual handling under fair scheduling.

External assumptions. Beyond these code-level boundaries, the mechanization relies on three external assumptions stated in sections 4.2 and 6: (i) IVC/SNARK soundness — no adversary can produce a valid proof for a false statement; (ii) S-BAC shard honesty — fewer than one-third Byzantine validators per shard, consistent with the Chainspace fault model [Al-Bassam et al.(2018)]; and (iii) network fairness — weak fairness for commit and abort actions, eventual message delivery for cross-shard coordination. These are the irreducible trust assumptions of the system; every safety theorem is conditional on (i), and every liveness claim is conditional on (ii) and (iii).

7.6 Limitations

The limitations fall into two categories: *trust boundaries* (items 1, 3, 4) mark where the mechanization ends and external assumptions begin; *specification gaps* (items 2, 5, 6) mark places where a stronger formalization is feasible and planned.

1. **The ZK verifier is an oracle, not a theorem.** The predicate `allProofsVerified` checks phase flags set by an external system. Every safety theorem is conditional: "if the verifier is sound, then …" Breaking the verifier breaks the entire chain of trust from protocol compliance to committed state. The model proves the *conditional* correctly; the *antecedent* is a cryptographic assumption that lies outside the formalization.

2. **Phase ordering is not statically enforced.** The eight `Step` constructors do not prevent `commit` from firing before `lockInputs`, or `handleE` before `installH`. The `ValidTxTransition` and `ValidProofTransition` inductives define the correct sequences and prove reachability (`committing_reachable`, `verified_reachable`), but they are not wired into the step guards. An indexed inductive family parameterized by the current phase would close this gap.

3. **Liveness is argued, not proved.** The mechanization proves the supporting lemmas — every pending transaction has at least one enabled step (`pending_can_step`), abort is always available, `handleEffect` strictly decreases the effect queue — but the temporal conclusions (eventual commit, eventual termination, eventual effect handling) require fairness assumptions that cannot be expressed in Lean's logic. These are validated by TLC model checking on bounded instances.

4. **The S-BAC protocol is a specification, not a verified implementation.** `SBAC.lean` defines the *logical predicates* a correct S-BAC implementation must satisfy and proves the

bidirectional theorems relating global and local validity. It does not model message passing, quorum voting, or Byzantine fault tolerance. The protocol's correctness is assumed from the established literature [Al-Bassam et al.(2018)]; mechanizing this layer — perhaps using a framework like Verdi's verified system transformers [Wilcox et al.(2015)] — would close the gap between the mechanized proofs and a deployed system.

5. **No programming-language semantics.** The model formalizes scripts, traces, and event structures but not the programming surface — no coroutine calculus, no channel types, no effect-handler operational semantics. The gap between what a programmer writes and what the model reasons about is bridged only by the design document.

6. **All identifiers are `Nat`.** `UTXOId`, `TxId`, `InterfaceId`, `ProcessId`, `RoleId`, `EventId`, and `CommitmentHash` are all `abbrev Nat`. This collapses the type distinction between fundamentally different domains: a transaction ID can typecheck where a UTxO ID is expected. The proofs remain sound — they never rely on type-level distinction — but the specification is weaker than it could be against modeling errors. Refactoring to opaque wrapper types is planned.

8 Related Work

ICE-UTxO sits at the intersection of multiparty session types, UTxO-based blockchain models, sharded consensus, and mechanized verification of concurrent systems. We organise the related literature along these four axes, devoting the most space to session-type theory, which is closest to our contribution. Table 6 summarizes the relationship between each ICE-UTxO component and its prior art.

8.1 Multiparty Session Types

Foundational theory. The multiparty session type (MPST) discipline originates with Honda, Yoshida, and Carbone [Honda et al.(2016)], who introduced *global types* as choreographic specifications from which local endpoint types are obtained by *projection*. Their framework guarantees communication safety, session fidelity, and progress for asynchronous sessions with arbitrarily many participants. ICE-UTxO adopts the global-type/projection architecture wholesale but departs in two critical ways: (i) our global types are indexed by *UTxO references* rather than channel names, so that each protocol step is anchored to an on-chain datum; and (ii) projection targets *transaction validator scripts* rather than process calculi, yielding proof obligations that are discharged at ledger-submission time rather than at compile time.

Hüttel et al. [Hüttel et al.(2016)] survey session types and behavioural contracts, classifying systems by their type discipline, communication medium, and verification strategy. In the taxonomy they propose, ICE-UTxO occupies a novel cell: it is an *asynchronous, event-structure-based* system whose communication medium is a *shared ledger* rather than point-to-point channels or shared memory.

Interleaving and global progress. Bettini et al. [Bettini et al.(2008)] study *global progress* for multiparty sessions that are dynamically interleaved on shared channels. Their key insight is that progress requires not only local typing but also a global ordering condition on channel usage. Coppo et al. [Coppo et al.(2013), Coppo et al.(2016)] subsequently show that this ordering can be *inferred* rather than annotated, using a constraint-based analysis.

ICE-UTxO faces a structurally analogous problem: multiple protocol instances may share UTxO outputs, creating potential deadlocks at the ledger level. Our solution encodes the global ordering directly in the *event-structure causal order* (section 3) and enforces it via S-BAC lock acquisition (section 6). The key difference is that in our setting the "channels" are UTxO

Table 6: Novelty positioning: each ICE-UTxO component, its prior art, and what is specifically new.

Component	**Prior Art**	**ICE-UTxO Contribution**
Coroutine frames on UTxOs	Ergo multi-stage contracts [Chepurnoy and Saxena(2019)]; Sui PTBs [Blackshear et al.(2023)]; PARSEC VM [PARSEC(2023)]	Atomic intra-transaction interleaving of multiple UTXO scripts with saved frame state $(pc, locals, methodId, hash)$ within the eUTxO model
Algebraic effects	Plotkin & Pretnar [Plotkin and Pretnar(2009)] in PL theory	First application of full algebraic effects (raise/handle/resume) to on-chain UTXO validators, with dynamically-scoped handlers bounded by transaction scope
MPST coordination	Honda et al. [Honda et al.(2016)]; Nomos [Das et al.(2021)] for binary session types in smart contracts	Extension to multiparty global types with event-structure semantics, projected to per-role local types verified at ledger submission time
PTB compilation	Sui PTBs [Blackshear et al.(2023)]	Formal compilation from MPST global types to PTB-style bytecode with mechanized correctness proofs
IVC proof-gating	ZEXE [Bowe et al.(2020)]/Aleo [Aleo(2021)] for proof-carrying txs; Nova [Kothapalli et al.(2022)]/HyperNova [Kothapalli and Setty(2024)] for IVC	Use IVC to verify multiparty protocol compliance (session fidelity), not just data integrity, with application-level privacy
S-BAC cross-shard	Chainspace [Al-Bassam et al.(2018)], OmniLedger [Kokoris-Kogias et al.(2018)]	Composition of S-BAC with session-type witness verification, enforcing protocol structure as well as ledger invariants

cells whose consumption is *atomic and irrevocable*, which simplifies the progress argument but introduces the need for cross-shard coordination that classical MPST avoids.

Beyond duality. Scalas and Yoshida [Scalas and Yoshida(2019)] generalise binary session types beyond syntactic duality using a *rely/guarantee* formulation inspired by concurrent separation logic. ICE-UTxO draws on this philosophy when it decomposes a global protocol into per-validator proof obligations (section 3.3): each validator's correctness proof *relies* on the structural guarantees of the UTxO model (uniqueness of consumption) and *guarantees* local adherence to the projected session type. Our setting is inherently multiparty, however, so the rely/guarantee decomposition is driven by projection from a global type rather than by binary co-typing.

Event-structure semantics. Castellani, Dezani-Ciancaglini, and Giannini [Castellani et al.(2023)] develop an *event-structure semantics* for asynchronous multiparty session types, replacing the traditional operational semantics based on labelled transition systems with a denotational model in which communication actions are events related by causality and conflict. This is closely related work to ours. Their event structures capture the true concurrency inherent in asynchronous multiparty interaction, avoiding the artificial interleaving that LTS-based semantics impose.

ICE-UTxO builds directly on this line of work. Our *transaction event structures* (section 3) instantiate the Castellani–Dezani-Ciancaglini–Giannini framework in a blockchain setting: events are ledger transitions, causality is the UTxO spend relation, and conflict arises when two transactions attempt to consume the same output. We extend their model in three ways: (i) we add *algebraic effects* to events, allowing each transaction to carry computational side-effects interpreted by the ledger runtime; (ii) we annotate events with *proof witnesses*, transforming the event structure into a *proof-carrying* artefact; and (iii) we provide a *mechanised* (Lean 4) equivalence proof between the event-structure semantics and the operational semantics, whereas Castellani et al.'s results are paper-based.

Monitoring and hybrid verification. Bocchi et al. [Bocchi et al.(2017)] introduce *monitors* derived from multiparty session types for verifying communication at runtime. Neykova et al. [Neykova et al.(2017)] extend this approach to *timed* multiparty sessions. ICE-UTxO adopts a conceptually similar runtime-verification architecture, but the "monitor" is the ledger's transaction-validation logic itself. Each transaction carries a proof witness checked by the validator script before admission to the chain, providing *enforcement* rather than mere detection.

Hu and Yoshida [Hu and Yoshida(2016)] propose *hybrid session verification*, combining static typing of the protocol skeleton with runtime checking of data-dependent branching. ICE-UTxO follows a broadly similar hybrid strategy: the *structural* properties of the protocol (ordering, participation, branching topology) are verified statically by our Lean 4 metatheory, while *payload* properties (datum validity, value conservation) are checked dynamically by on-chain validators using proof-carrying witnesses.

Tooling and code generation. The *Scribble* protocol description language [Yoshida et al.(2014)] provides a practical surface syntax for global types. *NuScr* [Yoshida et al.(2021)] extends Scribble with a CFSM-based backend supporting automated protocol validation and endpoint code generation. ICE-UTxO's protocol description layer (section 3) is inspired by Scribble's design but targets a different backend: instead of generating communicating finite-state machines, our compiler produces *Plutus validator scripts* annotated with proof obligations.

Voinea et al. [Voinea et al.(2020)] present *StMungo*, translating Scribble protocols into typestate specifications for Java. King et al. [King et al.(2019)] apply MPST to web development with static linearity. Miu et al. [Miu et al.(2021), Miu et al.(2020)] extend this line to TypeScript with WebSocket code generation. These tools demonstrate the practical viability of MPST-based code generation. ICE-UTxO contributes to this tradition by targeting *blockchain validator scripts*—a domain where code-generation correctness is especially critical because deployed validators cannot be patched and govern the movement of real assets.

Session types for smart contracts. Das, Balzer, Hoffmann, and Pfenning [Das et al.(2021)] introduced the Nomos system, which is the most directly relevant prior work applying session types to smart contract programming. Nomos uses binary session types grounded in intuitionistic linear logic to ensure that digital contracts satisfy resource-linearity properties: tokens cannot be duplicated or destroyed outside of explicitly authorized operations. Nomos contracts communicate via typed channels, and the type system statically guarantees that all protocol interactions conform to their session type specifications. However, Nomos differs from ICE-UTxO in several important respects. First, Nomos uses binary (two-party) session types, while ICE-UTxO uses multiparty session types with arbitrarily many roles—a distinction that matters for protocols involving more than two participants (e.g., the collateralized loan scenario with oracle, borrower, liquidator, and coordinator). Second, Nomos operates within its own custom contract language and runtime, while ICE-UTxO extends the existing eUTxO model and targets existing Plutus-style validators. Third, Nomos verifies protocol adherence at compile time through its type system, while ICE-UTxO verifies protocol adherence at ledger submission time through

IVC proof witnesses—a design choice driven by the open, permissionless nature of blockchain systems where contract code may come from untrusted sources. ICE-UTxO can be viewed as extending the Nomos vision from binary to multiparty, from a custom runtime to the eUTxO ledger, and from static type checking to dynamic proof-carrying verification.

Adaptation and replication. Harvey et al. [Harvey et al.(2021)] introduce *connection actions* that allow sessions to adapt at runtime by adding or removing participants. ICE-UTxO supports a limited form of adaptation through its *coroutine suspension* mechanism: a participant may suspend, allowing another party to resume in a later transaction. However, we do not currently support arbitrary participant addition or removal.

Le Brun et al. [Brun et al.(2025)] extend MPST with *replication*, enabling protocols that spawn unboundedly many sub-sessions. This is relevant because blockchain protocols often involve unbounded repetition (e.g., a payment channel processing arbitrarily many off-chain updates). Our current formalisation handles bounded unfolding; incorporating replication is future work.

Tirore et al. [Tirore et al.(2025)] provide the first *mechanised* proof of subject reduction for MPST, formalised in Coq. ICE-UTxO shares this commitment to mechanised metatheory but targets *ledger-level safety and progress* rather than subject reduction for a process calculus, and is carried out in Lean 4 rather than Coq. We view the two as complementary: their mechanised subject reduction could serve as a certified front-end feeding into our ledger-level back-end.

8.2 UTxO Models and Blockchain Verification

The UTxO model was introduced with Bitcoin [Nakamoto(2008)], where each transaction consumes outputs produced by previous transactions and creates new ones. Bitcoin's scripting language is intentionally limited, making transactions easy to validate in parallel—a property ICE-UTxO inherits—but precludes the expression of complex multi-step protocols.

Chakravarty et al. [Chakravarty et al.(2020)] introduce the *extended UTxO* (eUTxO) model underlying Cardano. eUTxO enriches Bitcoin's model with *datums* and *validator scripts*. ICE-UTxO extends eUTxO in three directions: (i) we add *coroutine state* to datums, enabling multi-step protocols encoded as suspended computations; (ii) we introduce *algebraic effects* into the validator execution model; and (iii) we require each transaction to carry a *proof witness* demonstrating conformance to a projected session type.

Melkonian et al. [Melkonian et al.(2019)] provide a mechanically verified reference semantics for the Cardano ledger in Agda. ICE-UTxO's Lean 4 formalisation covers analogous ground but extends the model with session-type indexing and proof-carrying transactions, and additionally proves the correspondence between event-structure and operational semantics.

Sui's *Move* language [Blackshear et al.(2019)] adopts an *object-centric* model with structural similarities to eUTxO: objects are analogous to UTxO outputs, and *programmable transaction blocks* [Blackshear et al.(2023)] allow multiple object operations to be composed atomically. Move's linear type system ensures resource safety at the language level. ICE-UTxO differs in employing multiparty session types for *protocol-level* safety—ensuring not only that individual resources are used correctly but that the overall multi-party interaction adheres to a choreographic specification.

Ergo [Chepurnoy and Saxena(2019)] extends the UTXO model with multi-stage contracts, where a sequence of transactions can be linked through transaction trees such that the output of one stage constrains the inputs of the next. This achieves a form of multi-step contract execution within the UTXO paradigm, but without atomicity across stages—each stage is a separate on-chain transaction, and intermediate states are globally visible. ICE-UTxO differs by enabling multiple execution steps within a single atomic transaction through its coroutine mechanism, and by providing formal session-type guarantees about the coordination pattern.

Table 7: Comparison of UTXO-family models across key dimensions.

	Multi-step	*Cross-contract*	*Algebraic effects*	*Proof-carrying*	*Formal verif.*	*Sharding*
Bitcoin	—	—	—	—	—	—
Ergo	✓*	partial	—	—	—	—
Cardano eUTxO	—	partial	—	—	✓†	—
Nervos CKB	partial	✓	—	—	—	—
Sui (object)	✓	✓	—	—	✓	✓
ICE-UTxO	✓	✓	✓	✓	✓	✓

*Multi-step across transactions (not atomic). †Agda formalization of ledger rules [Melkonian et al.(2019)].

Ergo's approach is more deployable today (it requires no ZK proofs or session types) but offers weaker guarantees about the correctness of multi-step interactions.

O'Connor [O'Connor(2017)] introduced Simplicity, a low-level, combinator-based smart contract language designed for Bitcoin-style UTXO chains with formal semantics defined in Coq. Simplicity prioritizes formal verifiability at the language level—every Simplicity program has a precise denotational semantics—but does not address multi-step coordination or cross-contract communication. ICE-UTxO operates at a higher level of abstraction, providing coordination primitives (coroutines, effects, session types) that could in principle be compiled to a Simplicity-like substrate.

Bartoletti and Zunino [Bartoletti and Zunino(2018)] introduced BitML, a process-algebra-based domain-specific language for Bitcoin smart contracts. BitML allows developers to specify contracts as concurrent processes with formal semantics, and a compiler translates BitML specifications into standard Bitcoin transactions. BitML shares ICE-UTxO's goal of bringing formal methods to UTXO contract development, but targets the much more constrained Bitcoin scripting environment and uses a different theoretical foundation (process algebra vs. session types with event structures).

The Nervos CKB [Nervos Network(2019)] generalizes the UTXO model through a *Cell* abstraction with generalized lock scripts and type scripts, providing a flexible substrate for programmable state. Like eUTxO, CKB cells carry persistent state and are consumed/produced atomically, but CKB's type scripts provide more flexible validation logic. ICE-UTxO's coroutine and session-type machinery could in principle be layered on top of CKB's Cell model as well as Cardano's eUTxO.

Möser, Eyal, and Sirer [Möser et al.(2016)] proposed Bitcoin covenants—UTXO extensions that enable a spent output to constrain the outputs of the spending transaction, propagating conditions forward in the UTXO graph. This is early work on adding expressiveness to UTXO outputs and is conceptually related to ICE-UTxO's frame propagation, where a coroutine's suspension state constrains how the output UTxO may be consumed in future steps.

8.3 Sharded Consensus

Al-Bassam et al. [Al-Bassam et al.(2018)] introduce *Chainspace*, a sharded smart-contract platform using *Sharded Byzantine Atomic Commit* (S-BAC) to execute transactions spanning multiple shards. ICE-UTxO adopts S-BAC as its cross-shard coordination primitive (section 6) but adds protocol-level verification: each sub-transaction carries a proof witness derived from the MPST projection, and the atomic-commit decision incorporates witness validation alongside the usual safety checks. This means ICE-UTxO's cross-shard transactions respect not only ledger-

level invariants (no double spending, value conservation) but also application-level protocol structure (session fidelity, progress).

Classical distributed-systems literature offers several relevant atomic-commit protocols. Two-phase commit (2PC) [Gray(1978)] provides atomicity but is blocking. Three-phase commit (3PC) [Skeen(1981)] eliminates the blocking window under crash failures but does not tolerate Byzantine faults. BFT atomic-commit protocols [Kokoris-Kogias et al.(2018)] extend atomic commit to adversarial settings, as does S-BAC. ICE-UTxO's contribution is orthogonal to the choice of commit protocol: we show how to *layer session-type verification on top of any atomic-commit primitive*, so that the commit decision reflects protocol-level correctness in addition to data-level consistency.

8.4 Mechanized Proofs of Concurrent Systems

IronFleet [Hawblitzel et al.(2015)] demonstrates end-to-end verification of practical distributed systems in Dafny, covering both a Paxos-based replicated state machine and a sharded key-value store. IronFleet decomposes verification into a *protocol layer* (modelled as a state machine) and an *implementation layer* (verified to faithfully implement the protocol). ICE-UTxO adopts a similar two-layer decomposition—our event-structure semantics serves as the protocol layer, and the UTxO operational semantics as the implementation layer—but differs in two respects. First, our protocol layer is derived from multiparty session types rather than hand-written state machines. Second, our mechanisation targets Lean 4, enabling us to exploit dependent types for proof terms that are themselves first-class data—a property we exploit when embedding proof witnesses into transactions.

Verdi [Wilcox et al.(2015)] provides a Coq framework for implementing and verifying distributed systems using *verified system transformers*. ICE-UTxO's architecture exhibits a loose analogue: our event-structure metatheory is proven under an idealised model, and the S-BAC layer provides the "transformer" that lifts these guarantees to a Byzantine fault-tolerant setting. However, we do not yet formally verify the S-BAC layer itself; doing so using Verdi-style transformers is a natural direction for future work.

Aneris [Krogh-Jespersen et al.(2020)] builds on the Iris separation-logic framework in Coq to provide a program logic for reasoning about distributed systems. ICE-UTxO's concurrency is mediated entirely by the ledger (UTxO consumption is the sole synchronisation primitive), so we do not require Iris's fine-grained local-concurrency reasoning. On the other hand, Aneris's protocol logic for network communication could complement our work if extended to model ledger-mediated interaction.

Across all three systems, the verification target is a *general-purpose* distributed system. ICE-UTxO exploits the specific structure of UTxO ledgers (unique consumption, deterministic validation, immutable history) to obtain a simpler and more automated proof strategy. Our Lean 4 development completes with zero uses of `sorry` or unverified axioms, and the entire metatheory is constructive, meaning that proof witnesses can be extracted and embedded in transactions as first-class data. This *proof-carrying* property is the central novelty relative to the mechanised-verification literature: rather than verifying a system implementation once and trusting it thereafter, we verify each individual protocol step at the moment it is submitted to the ledger.

8.5 Proof-Carrying Transactions and IVC

The idea that programs or data can carry machine-checkable proofs of their own correctness originates with Necula's proof-carrying code, where executables carry proofs of safety properties that are verified before execution. ICE-UTxO applies this principle to the blockchain setting: transactions carry proofs of protocol compliance that are verified before ledger admission.

ZEXE [Bowe et al.(2020)] is the foundational work on proof-carrying transactions for blockchains. ZEXE introduced the idea that each transaction in a decentralized ledger can carry a zero-knowledge proof certifying that the state transition it represents satisfies application-specific predicates, without revealing the predicate itself or the witness data. Aleo [Aleo(2021)] deploys this idea in production, using a record-based model (structurally similar to UTXO) where each record transition is accompanied by a SNARK proof. Mina [Bonneau et al.(2020)] takes a different approach, using recursive SNARK composition to maintain a constant-size blockchain where the entire chain state is certified by a single proof.

Nova [Kothapalli et al.(2022)] and HyperNova [Kothapalli and Setty(2024)] are the IVC folding schemes that make incremental proof generation practical. Nova introduced the concept of folding—combining multiple constraint-satisfaction instances into one without a full SNARK proof at each step—enabling IVC with dramatically lower per-step proving costs. This is directly relevant to ICE-UTxO's proof architecture: each coroutine yield/resume step can be folded into the running IVC accumulator, and only the final proof needs full verification.

ICE-UTxO's use of proofs differs from these prior systems in a specific way. In ZEXE and Aleo, proofs certify that an arbitrary user-defined predicate was satisfied during a state transition—the proof system is general-purpose. In ICE-UTxO, proofs specifically certify that a multiparty session protocol was followed correctly: that the schedule of coroutine operations respected the causal order and conflict relations specified by the coordination script. This is a more structured use of proof-carrying data, where the "predicate" is not arbitrary but is derived from the MPST global type. The advantage is that the proof structure mirrors the protocol structure, enabling shard-local verification (each shard checks its local projection) and composability (protocol compliance can be checked independently of payload validation).

9 Discussion and Future Work

The ICE-UTxO model and its Lean mechanization establish a formal foundation for proof-carrying transactions with multiparty coordination. Several important directions remain open. They fall into three categories: *formalization gaps* (full language semantics, IVC circuit soundness, liveness, phase enforcement, type-safe identifiers, conflict closure), *practical deployment* (performance, resource accounting, real-world integration, developer ergonomics), and *system design* (MPST expressiveness trade-offs, abort and incentive compatibility).

Full language semantics. The current formalization captures the *ledger-level footprint* of coroutines and effects. The computational semantics of coroutine programs—instruction execution, stack management, yield points—remain to be formalized. A full language semantics would define the operational behavior of coroutine bodies and prove that well-typed programs produce effects consistent with their coordination scripts.

IVC circuit soundness. The predicate `allProofsVerified` is an oracle. Full end-to-end verification requires mechanizing the ZK arithmetization (e.g., PLONK/Halo2) and proving a linkage theorem: the circuit constraints imply the protocol predicates checked by the ledger.

Liveness and progress. Section 4.6 establishes conditional liveness under fairness assumptions with mechanized support lemmas; the liveness theorems are paper-level arguments for which TLC model checking found no counterexamples in bounded state spaces. Full mechanization of temporal liveness in Lean 4—requiring either coinductive trace reasoning or a temporal logic embedding—remains future work.

Computational extraction. All predicates are `Prop`-valued. `Bool`-valued versions of key predicates (e.g., `allProofsVerifiedBool`, `conflictsBool`, `isAcyclicBool`) with proven equivalence to their `Prop` counterparts would enable executable validators and property-based testing within Lean.

Resource accounting. Resource accounting (gas, compute budgets, memory limits) is outside the current scope. Adding a fuel argument to the `Step` relation—e.g., $\mathit{Step} : \mathit{Mode} \to \mathbb{N} \to \mathit{Ledger} \to \mathit{Ledger} \to \text{Prop}$—would formalize resource exhaustion as an abort condition without affecting the existing safety theorems.

Complexity and developer ergonomics. The ICE-UTxO architecture layers six distinct technical concepts—coroutines, algebraic effects, multiparty session types, PTB compilation, IVC proofs, and sharded atomic commit—each carrying its own learning curve and implementation complexity. That is a steep barrier to entry. But it is a deliberate design choice: each layer addresses a specific concern, and the layered architecture allows each to be understood, implemented, and verified independently.

The combined complexity is a legitimate concern for adoption. The conservative extension property provides one mitigation: developers who do not need multi-step coordination can ignore all six layers entirely, and their transactions behave exactly as standard eUTxO transactions. For developers who do need coordination, practical adoption will require high-level domain-specific languages that hide the formal machinery—a Scribble-like [Yoshida et al.(2014)] protocol description language with a compiler that generates the MPST global type, PTB program, and IVC circuit constraints automatically. Production tooling would need IDE integration, error messages framed in application-domain terms, and automated testing support.

Performance considerations. Per-transaction SNARK proof generation remains computationally expensive. Even with Nova's folding approach [Kothapalli et al.(2022)], generating the final proof for a complex multi-step transaction may require hundreds of milliseconds to seconds on commodity hardware. Verification is fast (typically under 10ms), so on-chain costs are modest, but the proving burden falls on transaction authors and may be prohibitive for latency-sensitive applications.

Several mitigations are available or emerging: hardware acceleration via GPUs and specialized prover ASICs, batched proof generation across multiple transactions, and delegation of proving costs to sophisticated participants (market makers, protocol operators). Proof generation is embarrassingly parallel—different coroutine steps can be proved independently and then folded—so proving time scales with hardware investment rather than being an inherent bottleneck. Empirical benchmarking on representative coordination scenarios is needed and is left to future work.

MPST versus simpler coordination mechanisms. Full multiparty session types are the most expressive coordination mechanism ICE-UTxO offers, but many practical scenarios may not require this expressiveness. Simple two-party interactions (e.g., an atomic swap) need no coordination machinery at all. Slightly more complex scenarios (e.g., a three-step oracle query) could be handled by state machines or sequential workflow languages.

The justification for MPST arises in scenarios involving multiple independent parties with complex requirements: DeFi protocols with multiple liquidity pools and oracles, cross-protocol interactions where composability requires formal interface contracts, and high-stakes financial applications where formal deadlock freedom is worth the specification effort. The MPST framework also provides a natural specification language—the global type is a readable, formal description of the intended interaction—which is valuable independently of the verification benefits.

Abort, griefing, and incentive compatibility. The MPST framework guarantees deadlock freedom, but it does not guarantee that all parties will cooperate. A malicious or unresponsive party can refuse to execute its next step, causing the transaction to time out and abort. The abort constructor ensures this is always safe—no partial state changes are committed—but repeated aborts impose opportunity costs on honest participants who have locked UTxOs and generated proofs for a transaction that ultimately fails.

Incentive mechanisms such as stake bonds (collateral slashed on unjustified abort), reputation systems, or economic penalties could mitigate griefing attacks. The formal model could incorporate these by adding a cost function to the abort constructor, though formalizing incentive compatibility is a substantial research direction in its own right.

Real-world integration. Connecting ICE-UTxO to actual eUTxO implementations (Cardano's Plutus, or a Sui-style PTB runtime) requires bridging the gap between the idealized model and concrete blockchain constraints: gas limits, transaction size bounds, serialization formats, and network protocols. Concretely, coroutine frames would reside in UTxO datum fields, subject to datum size limits (currently $\sim$16 KB on Cardano). Resume execution must be resource-metered: each coroutine step consumes execution units, with a per-transaction budget. The coordination witness adds to the transaction's serialized size. These constraints bound the complexity of coordination scripts that can be used in practice but do not affect the formal model's safety guarantees.

Phase transition enforcement. The `Step` constructors do not enforce phase ordering. Tightening via an indexed inductive family `Step : TxPhase → Ledger → Ledger → Prop` is planned.

Type-safe identifiers. All identifier types are `Nat` aliases. Replacing with wrapper structures (e.g., `structure UTXOId where val : Nat`) would add compile-time type safety.

Conflict-downward closure. The current well-formedness rules do not enforce that conflict is *hereditary* (downward-closed under causality), as required by Winskel's original event structures. Adding this constraint would bring the model into full alignment with the event-structure literature.

Scaling the formalization. At about 4,500 lines, the formalization would benefit from closer integration with Mathlib. Several hand-proved lemmas duplicate existing Mathlib results, and the broad `import Mathlib` could be replaced with targeted imports to reduce compile times.

10 Conclusion

ICE-UTxO proves that the gap between single-shot UTxO validators and rich multi-step, multiparty interactions can be bridged without abandoning the UTxO model's core strengths: atomicity, parallelism, and deterministic validation. The key architectural insight—treating transactions as proof-carrying implementations of multiparty session protocols—decomposes a complex coordination problem into three independently verifiable layers, each addressing a distinct concern.

The Lean 4 mechanization (4,500 lines, zero `sorry`, zero custom axioms)[1] establishes three principal results. First, strong conflict serializability via a constructive bubble-sort proof: all conflict-respecting permutations of the committed history produce identical core state, enabling

[1]The Lean 4 source, TLA+ specifications, and build instructions are available at `https://github.com/chaslern/Starstream`. The development builds against Lean 4.27.0 and Mathlib 4.27.0.

shards to process non-conflicting transactions in any order with guaranteed agreement. Second, concurrent-to-serial refinement: the concurrent ledger with locks, pending transactions, and effect handlers is a faithful implementation of a serial specification where transactions commit one at a time. Third, bidirectional MPST-to-ledger bridge theorems: shard-local verification of projected traces is both necessary and sufficient for global protocol consistency, reducing a global property to a conjunction of local checks.

The constructive bubble-sort serializability proof is the first machine-checked proof of this property for a UTxO-based ledger with concurrent interleaving semantics: prior mechanized UTxO work [Melkonian et al.(2019)] covers ledger rules but not serializability, and prior MPST mechanization [Tirore et al.(2025)] covers subject reduction but not ledger-level properties. The formal guarantees hold under four explicit security assumptions—ZK verifier soundness, phase discipline, S-BAC Byzantine tolerance ($f < n/3$ per shard), and collision-resistant hashing—stated in section 3.1. Mechanizing IVC circuit soundness and liveness properties remain the most important open problems for closing the gap between the abstract model and a deployable system.

By making the transaction schedule an explicit, proof-carrying artifact, ICE-UTxO turns coordination from an implicit encoding challenge into a first-class, formally verified component of the ledger.

References

[Al-Bassam et al.(2018)] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. 2018. Chainspace: A Sharded Smart Contracts Platform. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[Aleo(2021)] Aleo. 2021. Aleo: A Platform for Private, Programmable Applications. Aleo whitepaper.

[Bartoletti and Zunino(2018)] Massimo Bartoletti and Roberto Zunino. 2018. BitML: A Calculus for Bitcoin Smart Contracts. In *ACM Conference on Computer and Communications Security (CCS)*.

[Bernstein et al.(1987)] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

[Bettini et al.(2008)] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2008. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR*.

[Blackshear et al.(2019)] Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, and Runtian Zhou. 2019. Move: A Language with Programmable Resources. Libra/Diem project technical paper.

[Blackshear et al.(2023)] Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, Brandon Williams, and Lu Zhang. 2023. Sui Lutris: A Blockchain Combining Broadcast and Consensus. arXiv:2310.18042

[Bocchi et al.(2017)] Laura Bocchi, Tzu-Chun Chen, Pierre-Malo Denielou, Kohei Honda, and Nobuko Yoshida. 2017. Monitoring Networks through Multiparty Session Types. *Theoretical Computer Science* 669 (2017), 33–58.

[Bonneau et al.(2020)] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. 2020. Mina: Decentralized Cryptocurrency at Scale. ePrint 2020/352.

[Bowe et al.(2020)] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. 2020. ZEXE: Enabling Decentralized Private Computation. In *IEEE Symposium on Security and Privacy (S&P)*. ePrint 2018/962.

[Brun et al.(2025)] Matthew Le Brun, Simon Fowler, and Ornela Dardha. 2025. Multiparty Session Types with a Bang!. In *ESOP (LNCS, Vol. 15695)*. `https://doi.org/10.1007/978-3-031-91121-7_6`

[Cardano Improvement Proposal(2022)] Cardano Improvement Proposal. 2022. CIP-31: Reference Inputs. Cardano CIP.

[Castellani et al.(2023)] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. 2023. Event Structure Semantics for Multiparty Sessions. *Journal of Logical and Algebraic Methods in Programming* 131 (2023), 100844. `https://doi.org/10.1016/j.jlamp.2022.100844`

[Chakravarty et al.(2020)] Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. 2020. The Extended UTxO Model. In *Financial Cryptography Workshops*.

[Chepurnoy and Saxena(2019)] Alexander Chepurnoy and Amitabh Saxena. 2019. Multi-stage Contracts in the UTXO Model. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology (DPM/CBT) Workshop at ESORICS*.

[Coffman et al.(1971)] Edward G. Coffman, Michael J. Elphick, and Arie Shoshani. 1971. System Deadlocks. *Comput. Surveys* 3, 2 (1971), 67–78. `https://doi.org/10.1145/356586.356588`

[Coppo et al.(2013)] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. 2013. Inference of Global Progress Properties for Dynamically Interleaved Multiparty Sessions. In *COORDINATION*.

[Coppo et al.(2016)] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2016. Global Progress for Dynamically Interleaved Multiparty Sessions. *Mathematical Structures in Computer Science* 26, 2 (2016), 238–302.

[Das et al.(2021)] Ankush Das, Stephanie Balzer, Jan Hoffmann, and Frank Pfenning. 2021. Resource-Aware Session Types for Digital Contracts. In *IEEE Computer Security Foundations Symposium (CSF)*.

[Dwork et al.(1988)] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (1988), 288–323.

[Ergo Platform(2019)] Ergo Platform. 2019. Ergo: A Resilient Platform for Contractual Money. Ergo Platform whitepaper, v1.0.

[Gelashvili et al.(2023)] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. 2023. Block-STM: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. `https://doi.org/10.1145/3572848.3577501`

[Gray(1978)] Jim Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems: An Advanced Course*. Springer, 393–481.

[Harvey et al.(2021)] Paul Harvey, Simon Fowler, Sam Lindley, and Simon J. Gay. 2021. Multiparty Session Types for Safe Runtime Adaptation in an Actor Language. In *ECOOP (LIPIcs, Vol. 194)*.

[Hawblitzel et al.(2015)] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *SOSP*.

[Honda et al.(2016)] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (2016), 1–67.

[Hu and Yoshida(2016)] Raymond Hu and Nobuko Yoshida. 2016. Hybrid Session Verification through Endpoint API Generation. In *FASE*.

[Hüttel et al.(2016)] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. *Comput. Surveys* 49, 1 (2016), 3:1–3:36. `https://doi.org/10.1145/2873052`

[King et al.(2019)] Jonathan King, Nicholas Ng, and Nobuko Yoshida. 2019. Multiparty Session Type-Safe Web Development with Static Linearity. In *PLACES@ETAPS (EPTCS, Vol. 291)*.

[Kokoris-Kogias et al.(2018)] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *IEEE S&P*.

[Kothapalli and Setty(2024)] Abhiram Kothapalli and Srinath Setty. 2024. HyperNova: Recursive Arguments for Customizable Constraint Systems. In *CRYPTO (LNCS)*. 345–379. `https://doi.org/10.1007/978-3-031-68403-6_11` ePrint 2023/573.

[Kothapalli et al.(2022)] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. 2022. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. In *CRYPTO*.

[Krogh-Jespersen et al.(2020)] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *ESOP*.

[Kwon and Buchman(2016)] Jae Kwon and Ethan Buchman. 2016. Cosmos: A Network of Distributed Ledgers. `https://v1.cosmos.network/resources/whitepaper`.

[Lamport(1994)] Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems* 16, 3 (1994), 872–923.

[Mazurkiewicz(1987)] Antoni Mazurkiewicz. 1987. Trace Theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*. Lecture Notes in Computer Science, Vol. 255. Springer, 279–324. `https://doi.org/10.1007/3-540-17906-2_30`

[Melkonian et al.(2019)] Orestis Melkonian, Wouter Swierstra, and Manuel M. T. Chakravarty. 2019. Formal Investigation of the Extended UTxO Model. In *Workshop on Type-Driven Development (TyDe)*.

[Miu et al.(2020)] Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. 2020. Generating Interactive WebSocket Applications in TypeScript. In *PLACES@ETAPS (EPTCS, Vol. 314)*.

[Miu et al.(2021)] Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. 2021. Communication-Safe Web Programming in TypeScript with Routed Multiparty Session Types. In *CC*. https://doi.org/10.1145/3446804.3446854

[Möser et al.(2016)] Malte Möser, Ittay Eyal, and Emin Gün Sirer. 2016. Bitcoin Covenants. In *Financial Cryptography and Data Security (FC)*.

[Nakamoto(2008)] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System.

[Nervos Network(2019)] Nervos Network. 2019. CKB: A Common Knowledge Base for Crypto-Economy. Nervos Network whitepaper.

[Neykova et al.(2017)] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. 2017. Timed Runtime Monitoring for Multiparty Conversations. *Formal Aspects of Computing* 29, 5 (2017), 877–910.

[O'Connor(2017)] Russell O'Connor. 2017. Simplicity: A New Language for Blockchains. In *PLAS*.

[Papadimitriou(1986)] Christos H. Papadimitriou. 1986. *The Theory of Database Concurrency Control*. Computer Science Press.

[PARSEC(2023)] PARSEC. 2023. Coroutine-Based Smart Contract Virtual Machine. PARSEC VM whitepaper.

[Plotkin and Pretnar(2009)] Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *ESOP*.

[Scalas and Yoshida(2016)] Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *ECOOP (LIPIcs, Vol. 56)*.

[Scalas and Yoshida(2019)] Alceste Scalas and Nobuko Yoshida. 2019. Less is More: Multiparty Session Types Revisited. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 30.

[Skeen(1981)] Dale Skeen. 1981. Nonblocking Commit Protocols. In *SIGMOD*.

[Tirore et al.(2025)] Dawit Tirore, Jesper Bengtson, and Marco Carbone. 2025. Multiparty Asynchronous Session Types: A Mechanised Proof of Subject Reduction. In *ECOOP (LIPIcs, Vol. 333)*.

[Vinogradova and Melkonian(2024)] Polina Vinogradova and Orestis Melkonian. 2024. Message-Passing in the Extended UTxO Ledger. In *Workshop on Trusted Smart Contracts (WTSC), Financial Cryptography Workshops (LNCS, Vol. 14746)*. 150–169. https://doi.org/10.1007/978-3-031-69231-4_11

[Voinea et al.(2020)] A. Laura Voinea, Ornela Dardha, and Simon J. Gay. 2020. Typechecking Java Protocols with StMungo. In *FORTE*.

[Wilcox et al.(2015)] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, and Xi Wang. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *PLDI*.

[Winskel(1987)] Glynn Winskel. 1987. Event Structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*. Lecture Notes in Computer Science, Vol. 255. Springer, 325–392.

[Yakovenko(2018)] Anatoly Yakovenko. 2018. Solana: A New Architecture for a High Performance Blockchain. Solana Labs Whitepaper, v0.8.13.

[Yoshida et al.(2014)] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. 2014. The Scribble Protocol Language. In *TGC*.

[Yoshida et al.(2021)] Nobuko Yoshida, Fangyi Zhou, Francisco Ferreira, Raymond Hu, and Rumyana Neykova. 2021. Communicating Finite State Machines and an Extensible Toolchain for Multiparty Session Types. In *FCT (LNCS, Vol. 12867)*.

A Lean 4 Theorem Statements

This appendix reproduces the type signatures of thirteen key theorems from the ICE-UTxO Lean 4 mechanization. All theorems are fully proved with zero `sorry` and zero custom axioms.

Reading Lean signatures. Each `theorem` declaration has the form `theorem name (params) (hypotheses) : conclusion`. Parenthesized arguments `(x : Type)` are explicit parameters; hypotheses named `h…` are assumptions. The colon separates hypotheses from the conclusion (the property being proved). `List Tx` is a list of transactions, `Finset UTXOId` is a finite set of UTxO identifiers, and $\to$ denotes implication. The $\neg$ symbol is negation. No Lean programming knowledge is needed to read these signatures as mathematical statements.

Ledger Semantics (Track 1: StarstreamPilot.lean). Theorems A.1–A.6 concern the concurrent ledger model: conflict serializability, proof-gated commit, invariant preservation, and the concurrent-to-serial refinement. These results establish that the ICE-UTxO ledger satisfies standard safety properties under arbitrary interleaving of the eight `Step` constructors.

A.1 Core Commutativity

Non-conflicting transactions commute at the core state level. This is the algebraic heart of the serializability argument.

```
-- StarstreamPilot.lean:491
theorem core_commute (s : CoreState) (t1 t2 : Tx)
    (hnc : ¬ fullConflicts t1 t2) :
    applyCoreCommit (applyCoreCommit s t1) t2 =
    applyCoreCommit (applyCoreCommit s t2) t1
```

A.2 Strong Serializability via Bubble-Sort

Acyclic full-conflict precedence graphs imply all conflict-respecting permutations produce the same core state. This is the universal diamond property.

```
-- StarstreamPilot.lean:720
theorem acyclic_strong_serializable ( s : CoreState)
    (hist : List Tx)
    (hnodup : hist.Nodup)
    (hacyc : fullPrecGraphAcyclic hist) :
    strongCoreSerializable  s hist
```

A.3 Proof-Gated Commit

Any step that extends the ledger history must have verified proof commitments. Established by exhaustive case analysis on the 8-constructor `Step` inductive.

```
-- StarstreamPilot.lean:1074
theorem commit_requires_proof {m l l' tx} :
    Step m l l' →
    l'.history = l.history ++ [tx] →
    proofOk tx
```

A.4 Invariant Preservation

Every step of the operational semantics preserves the six-part ledger invariant. Floyd–Hoare style case analysis on all step constructors.

```
-- StarstreamPilot.lean:1321
theorem step_preserves_invariant {m : Mode}
    {l l' : Ledger}
    (hstep : Step m l l')
    (hinv : ledgerInvariant l) :
    ledgerInvariant l'
```

A.5 Concurrent-to-Serial Refinement

Every concurrent execution refines the serial specification via a stuttering simulation. Non-commit steps are stuttering; commit steps produce serial steps.

```
-- StarstreamPilot.lean:1480
theorem concurrent_refines_serial {m : Mode}
    {l  l : Ledger}
    (hexec : Steps m  l  l)
    (hinv : ledgerInvariant  l) :
    SerialSteps m (absLedger  l) (absLedger  l)
```

A.6 No Double-Spend Preservation

The no-double-spend invariant is preserved through commit operations, given that outputs are fresh and inputs are live.

```
-- StarstreamPilot.lean:1110
theorem commit_preserves_no_double_spend
    {l : Ledger} {tx : Tx}
    (hinv : noDoubleSpend l)
    (hfresh : outputsFresh l tx)
    (hlive : inputsLive l tx) :
    noDoubleSpend (applyCommit l tx)
```

Coordination Layer (Track 2: Script.lean). Theorems A.7–A.8 concern the MPST coordination layer: projection from global to local scripts and reconstruction of global trace consistency from local conformance. These results ensure that the per-role, per-shard view of a protocol is consistent with the global specification.

A.7 Projection Preserves Local Conformance

Global conformance of a script on a trace implies local conformance for every role on the projected trace.

```
-- Script.lean:880
theorem proj_localConform_of_globalConform
    (s : Script) (r : RoleId) (tr : List EventId)
    (h : s.globalConform tr) :
    s.localConform r (s.traceProj r tr)
```

A.8 Cross-Role Trace Reconstruction

Local conformance for all roles, combined with cross-role consistency, implies global trace consistency. This is the bottom-up direction of the MPST-to-ledger bridge.

```
-- Script.lean:615
theorem traceConsistent_of_local_and_cross
    (s : Script) (tr : List EventId)
    (hwf : s.wellFormed)
    (hnd : tr.Nodup)
    (hevents :  e, e  tr → e  s.events)
    (hlocal :  r  s.roles,
      Script.localConform s r (s.traceProj r tr))
    (hcross : s.crossRoleConsistent tr) :
    s.traceConsistent tr
```

PTB Compilation (PTB.lean). Theorems A.9–A.11 concern the PTB compilation layer: translation from PTB programs to event-structure scripts, validity of program-order traces, and construction of globally valid coordination witnesses. These results establish that the concrete PTB execution format correctly induces the event-structure semantics required by the coordination layer.

A.9 PTB-to-Script Well-Formedness

A PTB program with valid role assignments produces a well-formed event-structure script. Acyclicity of the induced order follows from the $i < j$ constraint.

```
-- PTB.lean:302
theorem Program.toScript_wellFormed
    (cfg : Config) (p : Program)
    (hroles : p.rolesOK cfg)
    (hkind : p.roleKindOK cfg) :
    (p.toScript cfg).wellFormed
```

A.10 Program-Order Trace Validity

The full program-order trace is a valid trace of the induced script when no conflicts exist. This is the base case for the PTB-as-proposed-trace argument.

```
-- PTB.lean:375
theorem Program.validTrace_trace
    (cfg : Config) (p : Program)
    (hno : p.noConflicts) :
    Script.validTrace (p.toScript cfg) (p.trace)
```

A.11 Witness Construction

A well-formed, conflict-free PTB program produces a globally valid coordination witness, combining script well-formedness with trace validity.

```
-- PTB.lean:595
theorem Program.witnessGlobalOK_of
    (cfg : AccessConfig) (p : Program)
    (keep : Nat → Bool)
    (hroles : p.rolesOK cfg.toConfig)
    (hkind : p.roleKindOK cfg.toConfig)
    (hconf : p.conflictFree keep)
    (hdown : p.downClosed keep) :
    witnessGlobalOK (Program.toWitness cfg p keep)
```

S-BAC Bridge (SBAC.lean). Theorems A.12–A.13 connect Track 1 and Track 2 through the coordination witness. Together they establish a bidirectional bridge: global witness validity decomposes into local validity (top-down), and local validity with consistency reconstructs global validity (bottom-up).

A.12 Global Witness Implies Local

A globally valid coordination witness projects to valid local witnesses for all roles. This is the top-down direction enabling S-BAC shard-local verification.

```
-- SBAC.lean:36
theorem witnessLocalOK_of_global
    (w : CoordWitness)
    (h : witnessGlobalOK w) :
    witnessLocalOK w
```

A.13 Consistent Witness Implies Global

A well-formed script with a consistent witness trace implies global conformance. Combined with Theorem A.8, this closes the bidirectional bridge.

```
-- SBAC.lean:41
theorem witnessGlobalOK_of_local_and_consistent
    (w : CoordWitness)
    (hwf : w.script.wellFormed)
    (hcons : witnessConsistent w) :
    witnessGlobalOK w
```

Table 8: Summary of mechanization metrics.

Metric	**Value**
Total lines of Lean 4	4,502
Declarations	∼303
Theorems	∼95
Lemmas	∼49
Definitions	∼159
Instances	0
Examples	0
Sorry count	0
Custom axioms	0
`Classical.choice`	0
`classical` tactic uses	1 (`Script.lean`:622)
Standard axioms	`propext`, `Quot.sound`, `funext`