VerIso: Verifiable Isolation Guarantees for Database Transactions

Shabnam Ghasemirad
ETH Zurich
Switzerland

Si Liu
ETH Zurich
Switzerland

Luca Multazzu
ETH Zurich
Switzerland

Christoph Sprenger
ETH Zurich
Switzerland

David Basin
ETH Zurich
Switzerland

ABSTRACT

Isolation bugs, stemming especially from design-level defects, have been repeatedly found in carefully designed and extensively tested production databases over decades. In parallel, various frameworks for modeling database transactions and reasoning about their isolation guarantees have been developed. What is missing however is a mathematically *rigorous* and *systematic* framework with tool support for formally verifying a wide range of such guarantees for *all* possible system behaviors.

We present VerIso, the first mechanized formal framework of its kind, developed within the theorem prover Isabelle/HOL. To showcase its use in verification, we model and verify two concurrency control protocols: the strict two-phase locking protocol satisfying strict serializability and a novel transaction protocol Eiger-PORT+ satisfying transactional causal consistency with convergence. Eiger-PORT+ is also of independent interest as it offers a stronger isolation guarantee and higher performance than the state-of-the-art. Moreover, we show how VerIso helps identify isolation bugs at the design stage. We derive known and new counterexamples for the TAPIR protocol from failed attempts to prove its claimed strict serializability; in particular, we show that it violates atomic visibility.

PVLDB Reference Format:
Shabnam Ghasemirad, Si Liu, Luca Multazzu, Christoph Sprenger, and David Basin. VerIso: Verifiable Isolation Guarantees for Database Transactions. PVLDB, 1(1): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ShabnamRad/VerIso

1 INTRODUCTION

Over the past decades, significant efforts have been devoted to developing reliable, high-performance databases. Starting from the centralized relational databases of the 1980s, development moved on to geo-replicated NoSQL key-value stores such as Dynamo and Cassandra, prioritizing availability over consistency. This progress continued with the emergence of NewSQL databases, which bridge the gap between SQL and NoSQL by supporting ACID transactions for better data integrity, while still being highly performant.

Proceedings of the VLDB Endowment, Vol. 1, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

On the theoretical side, this trend has led to the study of new isolation levels, accompanied by novel concurrency control mechanisms and database designs balancing isolation guarantees and system performance. In addition to the gold standard Serializability (SER) as well as the widely adopted Snapshot Isolation (SI) [24, 28, 46], many weaker isolation levels have emerged to cater for real-world applications. These include Read Atomicity (RA), supported by [6, 35, 36] and recently layered atop Facebook's TAO [13]; Transactional Causal Consistency with convergence (TCCv), supported by numerous academic transaction protocols [3, 10, 17, 36, 40, 41, 44, 53], as well as its variants, recently adopted by commercial databases such as Azure Cosmos DB [45], Neo4j [47], and ElectricSQL [20]; and the SI variants like Parallel SI (PSI) [52].

Unfortunately, isolation bugs have repeatedly manifested themselves in carefully designed and heavily tested databases [9, 18, 25–27]. In particular, all these databases have exhibited isolation bugs stemming from design-level defects, rather than pure implementation errors. For example, SER violations were found in YugabyteDB as its conflict-detection mechanism failed to account for read conflicts [29] and MariaDB's Galera cluster incorrectly claimed its support for SI [25, 33]. Lu et al. [42] recently reported another example of a design defect, where a timestamp inversion violates the claimed Strict Serializability (SSER) in TAPIR [60] and DrTM [57]. Eliminating such bugs, which requires costly redesigns, and having strong correctness guarantees are a crucial, challenging task for developers. In this situation, the formal verification of database designs is highly desirable, preferably in an early design stage. Indeed, the industrial adoption of formal methods in the wider area of production databases [5, 16, 48, 55] has recently become increasingly prevalent as a proactive approach to address such design errors.

The formal verification of concurrency control protocols necessitates a formal semantics for isolation guarantees. Many semantics have been proposed [1, 8, 9, 12, 15, 38, 58], including the first characterization of SER via dependency graphs [8], an axiomatic framework based on abstract executions [12], and the recent operational semantics for isolation guarantees [58]. Nonetheless, protocol verification based on these semantics relies on either (i) manual pen-and-paper proofs, which are error-prone for complex protocols, (ii) on testing, which can only check executions individually, or (iii) on model checking, which is limited to a small number of processes and transactions, and can therefore find anomalies, but not establish correctness. Hence, these approaches can miss design errors that violate the claimed isolation guarantees. For example, TAPIR violates SSER, despite its authors' formal modeling as well as both manual and model checking efforts [60].

Our main objective is thus to develop a *systematic* and mathematically *rigorous*, tool-supported specification and verification framework for concurrency control protocols and their isolation guarantees. Systematic means covering a wide range of isolation guarantees in a unified manner. Rigor is guaranteed by mathematical proofs showing that *all* possible database behaviors satisfy the desired isolation guarantee, irrespective of how many processes and transactions are involved.

To achieve this goal, we have developed VerIso, the *first mechanized* framework for *rigorously* verifying isolation guarantees of transaction protocols. VerIso is based on the centralized operational model of transactions proposed in [58], instantiatable by a range of prevalent isolation criteria, including SSER, SI, and TCCv. We formalize this model, called the *abstract transaction model* for short, in Isabelle/HOL [49], an interactive theorem prover for higher-order logic, which offers the expressiveness and strong soundness guarantees required for systematic and rigorous tool support.

To develop VerIso, we also make several substantial enhancements to the original model, beyond its formalization, which underpin our protocol correctness proofs and their automation. First, we significantly extend the abstract model with an extensive library of hundreds of lemmas, including those stating properties of the abstract model's underlying concepts (e.g., version lists and reader sets) and its invariants (e.g., read atomicity). The resulting VerIso library is *protocol-independent* and provides strong support for protocol correctness proofs. Second, we set up Isabelle's automatic proof tools to apply many lemmas automatically, thereby increasing overall proof automation and reducing the user's proof burden.

Within VerIso, we formalize transaction protocol designs as transition systems and establish their isolation guarantees by refinement of the appropriate instance of the abstract model. For weak isolation levels, where write conflicts are allowed (e.g., in TCCv), an extra proof step to reorder the commits by their timestamps may be required to enable the refinement. The VerIso library also includes a collection of sound proof rules for this reordering technique and for refinement, which yield explicit and clear reachability-based correctness guarantees. We also identify a collection of generic protocol invariants that can support the refinement proofs of a wide range of protocols. VerIso thus offers rigorous tool support for verifying isolation guarantees of newly designed transaction protocols at an early stage. This mitigates the cost induced by redesign, implementation patching, and re-deployment of faulty designs.

To validate VerIso, we verify the isolation guarantees of two database designs. First, we model and verify Strict Two-Phase Locking (S2PL) (combined with Two-Phase Commit), a classical protocol used in realizing SSER. We use this case study to illustrate different aspects of VerIso including protocol specification, refinement mapping, and refinement proof using invariants. Second, we propose a novel distributed transaction protocol for read- and write-only transactions, called Eiger-PORT+, and use VerIso to verify its TCCv guarantee. This is a more complex protocol, featuring *timestamp-based* optimistic concurrency control, thus requiring refinement combined with reordering to verify its isolation guarantee.

Our Eiger-PORT+ protocol offers a *stronger isolation guarantee* and *higher performance* than Eiger-PORT [43], the state-of-the-art causally consistent transaction protocol providing pure TCC *without* data convergence. Eiger-PORT+ is therefore also of independent interest, as it improves upon the previously conjectured, strongest isolation level, namely TCC, for *performance-optimal* (i.e., non-blocking, one-round, with constant meta-data) read-only transactions in the presence of transactional writes [43].

Finally, we show how to use VerIso to systematically discover isolation bugs in database designs based on unprovable proof obligations, from which we construct concrete counterexamples. As an illustrative example, we use the TAPIR [60] distributed transaction protocol claiming to provide SSER. We not only (re)discover the issue with real-time ordering [42], but also uncover *fractured reads*, a violation of *atomic visibility* [6], where a transaction's updates are only partially observed by others.

Contributions. Overall, we make the following contributions.

- We develop VerIso, the first mechanized framework for formally specifying database designs and systematically verifying their isolation guarantees for *all* their behaviors. Compared to existing work, VerIso is mechanized, more expressive, covers more isolation levels, and offers much stronger correctness guarantees.
- We model and verify two protocols of different types in VerIso: a *lock-based* protocol satisfying SSER and a novel *timestamp-based* protocol Eiger-PORT+ meeting TCCv. In particular, Eiger-PORT+ improves the upper bound of achievable isolation levels, from the previously conjectured TCC to TCCv, for performance-optimal read-only transactions in the presence of transactional writes.
- We demonstrate how VerIso can be used to find isolation bugs of different kinds in database designs by examining failed proof obligations. In particular, in the case of TAPIR, we construct counterexamples illustrating a new violation of atomic visibility.

2 BACKGROUND

2.1 Isolation Levels

Distributed databases provide various isolation levels, depending on the desired system scalability and availability. As shown in Figure 1, VerIso supports well-known isolation levels such as SER and SI, as well as more recent guarantees such as RA and TCCv. We will briefly explain some of these isolation guarantees.

Read Atomicity (RA). This is also known as *atomic visibility*, requiring that all or none of a transaction's updates are observed by other transactions. It prohibits *fractured reads* anomalies, such as Carol only observing one direction of a new (bi-directional) friendship between Alice and Bob in a social network.

Transactional Causal Consistency (TCC). This isolation level requires that two causally related transactions appear to all client sessions in the same causal order [2, 51]. It prevents *causality violations*, such as Carol observing Bob's response to Alice's message without seeing the message itself.

TCC with Convergence (TCCv). With TCC, different clients may observe causally unrelated transactions in different orders. TCCv's *convergence* property prevents this by requiring these clients' views to converge to the same state [3, 39]. For example, this prevents confusion created by Alice and Bob independently posting "Let's meet at my place" in a road trip planner. In practice, most causally consistent databases provide convergence.

Strict Serializability (SSER). This strengthens SER by requiring that the sequential execution of transactions witnessing SER preserves the real-time order of (non-overlapping) transactions.

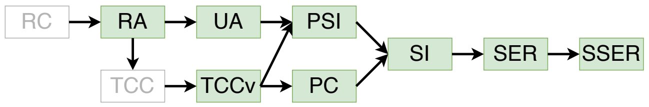

Figure 1: A hierarchy of prevalent isolation levels. $A \rightarrow B$ **means** A **is weaker than** B**. RC: read committed [7]; RA: read atomicity [6]; UA: update atomicity [12, 37]; TCC: transactional causal consistency [43]; TCCv: TCC with convergence [3, 40]; PC: prefix consistency [11]; SI: snapshot isolation [7]; PSI: parallel SI [52]; SER: serializability [50]; SSER: strict SER [50]. VerIso covers the isolation levels in green.**

2.2 Operational Semantics for Transactions

Xiong et al. [58] introduced a state-based operational semantics for atomic transactions that operate on distributed key-value stores (KVSs). This semantics is formulated as a labeled transition system (Section 2.3), called the *abstract transaction model* for short, which abstracts these KVSs into a single (centralized) multi-versioned KVS $\mathcal{K}$: key $\rightarrow$ list(version) that maps each key to a list of *versions*. Each version $\mathcal{K}(k, i)$ of a key k at list index i records (i) the value stored, (ii) the writer transaction, and (iii) the reader set, i.e., the set of transactions that have read this version. The reader set tracks write-read (WR) dependencies. The fact that, in a real, distributed system, each client *cl* has a different partial *client view* of $\mathcal{K}$ is modeled by explicitly representing these views in the model's configurations as mappings $\mathcal{U}(cl)$: key $\rightarrow \mathcal{P}(\mathbb{N})$, describing, for each key, the set of versions (denoted by list indices) visible to the client. Clients are assumed to process transactions sequentially.

The semantics assumes a *last-write-wins* conflict resolution policy and the *snapshot property*, ensuring that transactions read and write at most one version of each key. It also assumes that views are *atomic*, i.e., clients observe either all or none of a transaction's effects. These properties together ensure *atomic visibility* and establish Read Atomicity (RA) as the operational semantics' baseline isolation guarantee. Transactions are described by a *fingerprint* $\mathcal{F}$: key $\times$ {R,W} $\rightharpoonup$ value, which maps each key and operation (read or write) to a value, if any. The model has two types of transitions: *atomic commit*, which atomically executes and commits an entire transaction, and *view extension*, which monotonically extends a client's view of the key-value store. A commit transition can only be executed under certain conditions on the current KVS $\mathcal{K}$, the client's view $\mathcal{U}(cl)$, and the transaction's fingerprint $\mathcal{F}$. These conditions depend on the desired isolation guarantee. The framework is parameterized on these conditions and can be instantiated to eight different isolation guarantees (cf. the green boxes in Figure 1).

Note that the semantics' built-in atomic visibility and convergence (implied by the totally ordered version lists) precludes the representation of the TCC and RC isolation levels (cf. Figure 1). In future work, we plan to extend the model to also cover these levels.

2.3 Transition Systems and Refinement

We use *labeled transition systems* (LTS) to model database protocols and the abstract transaction model. An LTS $\mathcal{E} = (S, I, \{\xrightarrow{e} \mid e \in E\})$ consists of a set of states S, a non-empty set of initial states $I \subseteq S$, and a family of transition relations $\xrightarrow{e} \subseteq S \times S$, one for each event $e \in E$. We assume that our LTS models contain an idling event skip, defined by $s \xrightarrow{\text{skip}} s$, which we use in refinements (see below). We often define the relations $\xrightarrow{e}$ using *guard* predicates G_e and *update* functions U_e by $s \xrightarrow{e} s'$ if and only if $G_e(s) \wedge s' = U_e(s)$.

Example 1. An event dec to decrement a state variable x may be specified by dec : $x > 0 \wedge x' = x - 1$, where $G(x) = x > 0$ is the guard and $U(x) = x - 1$ is the update function. Primed variables refer to the value of x after the transition and variables not mentioned on the left of the update are assumed to be unchanged.

A state s is *reachable* if there is a sequence of transitions from an initial state leading to s. We denote the set of reachable states of $\mathcal{E}$ by reach$(\mathcal{E})$. A set of states I is an *invariant* if reach$(\mathcal{E}) \subseteq I$. Invariants are proved by showing that they hold in all initial states and are preserved by all state transitions.

Refinement relates two LTSs $\mathcal{E}_i = (S_i, I_i, \{\xrightarrow{e}_i \mid e \in E_i\})$, for $i \in \{1, 2\}$, which usually represent different abstractions levels. Given *refinement mappings* $r : S_2 \rightarrow S_1$ and $\pi : E_2 \rightarrow E_1$ between the LTSs' states and events, we say $\mathcal{E}_2$ refines $\mathcal{E}_1$, written $\mathcal{E}_2 \preccurlyeq_{r,\pi} \mathcal{E}_1$, if (i) $r(s) \in I_1$ for all $s \in I_2$ and (ii) $r(s) \xrightarrow{\pi(e)}_1 r(s')$ whenever $s \xrightarrow{e}_2 s'$. Using guards and updates, (ii) reduces to two proof obligations: assuming $G^2_e(s)$ prove (a) $G^1_{\pi(e)}(r(s))$ (*guard strengthening*) and (b) $r(U^2_e(s)) = U^1_{\pi(e)}(r(s))$ (*update correspondence*). Refinement guarantees the inclusion of sets of reachable states (modulo r), i.e., $r(\text{reach}(\mathcal{E}_2)) \subseteq \text{reach}(\mathcal{E}_1)$. Refinement proofs often require invariants of the concrete model to strengthen the refinement mapping.

Example 2. We implement the decrement event above by splitting it into two events that use a register t: (1) dec$_0$: $x > 0 \wedge t' = x - 1$ and (2) dec$_1$: $x \neq t \wedge x' = t$. We define $r((x, t)) = x$, $\pi(\text{dec}_0) = \text{skip}$, and $\pi(\text{dec}_1) = \text{dec}$, i.e., dec$_0$ refines the idling event, as it does not affect the abstract state, unlike dec$_1$, which refines dec. Supposing that initially $x \geq 0$ and $t = x$, condition (i) is clearly satisfied. For condition (ii), the proof obligations (a) and (b) are satisfied for dec$_0$ (skip's guard is true). For event dec$_1$, we must show (a) $x > 0$ and (b) $t = x - 1$, assuming $x \neq t$. Both are unprovable. However, we can prove that $x \neq t \Rightarrow x > 0 \wedge t = x - 1$ is an invariant of the concrete model and thus complete the refinement proof.

3 THE VERISO FORMAL FRAMEWORK

We first give an overview of VerIso and its application to the verification of concurrency control protocols (Section 3.1). We then present how we formalize the abstract transaction model [58] in Isabelle/HOL (Section 3.2). Finally, we discuss our protocol verification technique, including a technique for commuting independent events, which is sometimes required to complement refinement (Section 3.3), for example, for verifying Eiger-PORT+ (Section 5).

3.1 Overview and Notation

3.1.1 Abstract transaction model. We formalize the abstract model in Isabelle/HOL as an LTS $\mathcal{I}(\mathit{IL})$, parameterized by an isolation level $\mathit{IL} \in$ {RA, UA, TCCv, PSI, CP, SI, SER, SSER} (cf. Figure 1). The model's states $(\mathcal{K}, \mathcal{U})$, called configurations, consist of a central KVS $\mathcal{K}$ and the client views $\mathcal{U}$. The model has three events: an (atomic) commit event, whose guard depends on IL, a view extension event, and the idling event. We specify a protocol's desired isolation guarantee by the appropriate instance of the model $\mathcal{I}(\mathit{IL})$.

3.1.2 Protocol modeling. We consider distributed systems with one transaction coordinator per client and several shard/replica servers that handle the clients' reads and writes to keys. For simplicity, we integrate the coordinator into the client and assume that each server stores one key. We focus on transactions' concurrency control, and do not model their replication which we consider to be orthogonal.

We formalize protocol designs as LTS in Isabelle/HOL. This requires the definition of the LTS' global configurations and events. The former are composed of the clients' and the servers' local configurations. We only allow the events to change either a single client or a single server's configuration, which ensures that clients and servers can be seen as independent components with interleaved events. To exchange information, we allow these components to directly access each other's configurations. This is a standard abstraction in protocol modeling, which can later be refined into explicit message passing through a communication medium.

3.1.3 Basic protocol verification by refinement. We use refinement as the basic proof technique to establish that a protocol model $\mathcal{M}$ satisfies the desired isolation guarantee $\mathbb{IL}$, i.e.,

$$\mathcal{M} \preccurlyeq_{r,\pi} \mathcal{I}(\mathbb{IL}). \tag{1}$$

for suitable refinement mappings r and π on protocol states and events. In particular, we must identify one or more protocol events that refine the abstract commit event and prove for each of these (i) guard strengthening, i.e., that the protocol event's guards imply the premises of the commit transition rule and (ii) update correspondence, i.e., that abstracting the result of the protocol configuration's concrete update equals the result of the abstract update of the abstracted protocol configuration. Events that change a client's view of the key-value store (e.g., by making additional versions visible) must be similarly shown to refine the abstract view extension event, while all the remaining events must refine the abstract idling event. VerIso provides proof rules that support such refinements. Recall that the refinement (1) implies the following *correctness condition*:

$$r(\mathsf{reach}(\mathcal{M})) \subseteq \mathsf{reach}(\mathcal{I}(\mathbb{IL})), \tag{2}$$

where r is the state component of the refinement mapping.

3.1.4 Notation. To enhance readability, we use standard mathematical notation where possible and blur the distinction between types and sets. For a function $f : A \to B$, we denote by $f[x \mapsto y]$ the function that maps x to y and otherwise behaves like f. For a partial function $g : A \rightharpoonup B$, we write $dom(g)$ for the domain of g, and $g(x) = \bot$ if $x \notin dom(g)$. For a relation $R \subseteq A \times A$, R^{-1} denotes its converse and R^{+} its transitive closure. We display Isabelle code excerpts in grey boxes (Sections 4 and 5).

3.2 Formalizing the Abstract Transaction Model

We present our formalization of the abstract transaction model, leaving the full details to the supplementary material [21].

As mentioned above, we define the abstract model as an LTS $\mathcal{I}(\mathbb{IL})$, parameterized by an isolation level $\mathbb{IL}$. In particular, the commit event's guard depends on $\mathbb{IL}$, which is specified by two elements:

- a relation $R_{\mathbb{IL}} \subseteq \mathit{TxID} \times \mathit{TxID}$ on transaction identifiers, and
- a predicate $\mathsf{vShift}_{\mathbb{IL}}(\mathcal{K}, u, \mathcal{K}', u')$ on two KVSs and two views.

We will further describe $R_{\mathbb{IL}}$ and $\mathsf{vShift}_{\mathbb{IL}}$ below. Transaction identifiers $t_{sn}^{cl} \in \mathit{TxID}$ are indexed by the issuing client cl and a (monotonically increasing) sequence number sn.

The abstract commit event's transition relation is defined by:

$$\frac{\begin{array}{c}\mathcal{U}(cl) \sqsubseteq u \quad \mathsf{canCommit}(\mathcal{K}, u, \mathcal{F}, R_{\mathbb{IL}}) \quad \mathsf{vShift}_{\mathbb{IL}}(\mathcal{K}, u, \mathcal{K}', u') \\ \mathsf{consWithSnapshot}(\mathcal{K}, u, \mathcal{F}) \quad \mathsf{wf}(\mathcal{K}, u) \quad \mathsf{wf}(\mathcal{K}', u') \\ t_{sn}^{cl} \in \mathsf{nextTxids}(\mathcal{K}, cl) \quad \mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, t_{sn}^{cl}, u, \mathcal{F})\end{array}}{(\mathcal{K}, \mathcal{U}) \xrightarrow{\mathsf{commit}(cl, sn, u, \mathcal{F})}_{\mathbb{IL}} (\mathcal{K}', \mathcal{U}[cl \mapsto u'])}$$

The transition (below the bar) updates the configuration $(\mathcal{K}, \mathcal{U})$ to the new configuration $(\mathcal{K}', \mathcal{U}[cl \mapsto u'])$, where $\mathcal{K}'$ is the updated KVS and $\mathcal{U}[cl \mapsto u']$ updates the client cl's view to u'. Both $\mathcal{K}'$ and u' are determined by the rule's premises (above the bar), which act as the event's guards with the following meanings:

- $\mathcal{U}(cl) \sqsubseteq u$: This condition allows one to extend the client cl's current view to a (point-wise) larger one before committing.
- $\mathsf{canCommit}(\mathcal{K}, u, \mathcal{F}, R_{\mathbb{IL}})$: This is the central commit condition, which ensures that it is safe to commit a transaction at a given isolation level $\mathbb{IL}$. It requires that the set of visible transactions $\mathsf{visTx}(\mathcal{K}, u)$ (i.e., the writers of the versions that the view u points to) is *closed* under the relation $R_{\mathbb{IL}}$ in the sense that
$$(R_{\mathbb{IL}}^{-1})^{+}(\mathsf{visTx}(\mathcal{K}, u)) \subseteq \mathsf{visTx}(\mathcal{K}, u) \cup \mathsf{rdonly}(\mathcal{K}). \tag{3}$$
In other words, following the relation $R_{\mathbb{IL}}$ backwards from visible transactions, we only see visible or read-only transactions.
- $\mathsf{vShift}_{\mathbb{IL}}(\mathcal{K}, u, \mathcal{K}', u')$: This condition constrains the allowed modifications of the client view during the commit. The framework uses this predicate to capture session guarantees like monotonic reads and read-your-writes.
- $\mathsf{consWithSnapshot}(\mathcal{K}, u, \mathcal{F})$: This requires the consistency of the fingerprint $\mathcal{F}$'s reads with the latest values for the respective keys that are visible in the view u (most recent snapshot).
- $\mathsf{wf}(\mathcal{K}, u)$ and $\mathsf{wf}(\mathcal{K}, u')$: This requires that the views u and u' are *wellformed*, i.e., atomic and all pointed-to versions exist.
- $t_{sn}^{cl} \in \mathsf{nextTxids}(\mathcal{K}, cl)$: This condition obtains a fresh transaction ID t_{sn}^{cl}, where the sequence number sn is larger than any of the client cl's sequence numbers used in $\mathcal{K}$.
- $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, t_{sn}^{cl}, u, \mathcal{F})$: The KVS $\mathcal{K}'$ is obtained from $\mathcal{K}$ by adding the operations described by the fingerprint $\mathcal{F}$: the writes append a new version with writer ID t_{sn}^{cl} to the respective key's version list, and the reads add t_{sn}^{cl} to the respective versions' reader sets.

The view extension event is defined by the rule:

$$\frac{\mathcal{U}(cl) \sqsubseteq u \quad \mathsf{wf}(\mathcal{K}, u)}{(\mathcal{K}, \mathcal{U}) \xrightarrow{\mathsf{ext\_view}(cl, u)}_{\mathbb{IL}} (\mathcal{K}, \mathcal{U}[cl \mapsto u])}$$

This event simply extends a client cl's view from $\mathcal{U}(cl)$ to a well-formed view u. It abstractly models that additional versions of certain keys become visible to the client.

3.2.1 Instantiation to concrete isolation levels. To instantiate this abstract model for a particular isolation level $\mathbb{IL}$, one specifies the relation $R_{\mathbb{IL}}$ and the predicate $\mathsf{vShift}_{\mathbb{IL}}$. We describe here the definitions for $\mathbb{IL} \in \{\mathsf{TCCv}, \mathsf{SSER}\}$, which we use later and we refer the reader to [58] for details on other instantiations. For TCCv, we define $R_{\mathsf{TCCv}} = \mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}$, where SO captures the clients' session orders and $\mathsf{WR}_{\mathcal{K}}$ is the write-read dependency between transactions in $\mathcal{K}$, determined by the reader set associated to each version. The resulting commit condition $\mathsf{canCommit}_{\mathsf{TCCv}}$ requires that the views are closed under *causal dependencies*. The predicate $\mathsf{vShift}_{\mathsf{TCCv}}$ captures the monotonic reads and read-your-writes session guarantees. For SSER, we have $R_{\mathsf{SSER}} = \mathsf{WW}_{\mathcal{K}}^{-1}$, where $\mathsf{WW}_{\mathcal{K}}$ is the write-write dependency (i.e., per-key version order) on transactions in $\mathcal{K}$. Hence, the commit condition $\mathsf{canCommit}_{\mathsf{SSER}}$ expresses that the views must include *all* versions in the $\mathcal{K}$ before the commit. The condition $\mathsf{vShift}_{\mathsf{SSER}}$ is true, i.e., it always holds.

3.2.2 Discussion of formalization. The formalization involved some design choices and adaptations compared to [58]. First, to represent the framework's elements in Isabelle/HOL, we used Isabelle's type system to avoid proving invariants where possible. For example, we represent fingerprints as partial functions rather than relations. Second, the canCommit condition in Equation (3) is an equivalent reformulation of the original one. We find our version easier to understand and prove. VerIso's library provides proof rules that help establish this condition. Third, the paper [58] defines a KVS wellformedness condition including the snapshot property, and *assumes* it for all KVSs. In our formalization, we *proved* KVS wellformedness as an invariant of the abstract model.

3.3 Protocol Verification Technique

We first describe our approach to constructing refinement proofs. We then demonstrate that refinement alone is sometimes insufficient to prove the correctness condition (2) above. Finally, we introduce an additional general proof technique for those cases.

3.3.1 Constructing refinement proofs. After modeling a protocol, we have to define the refinement mappings r and π. We already covered the definition of π in Section 3.1.3. To define r, we must reconstruct an abstract configuration from a concrete protocol configuration s, i.e.,

$$r(s) = (\mathcal{K}\_\mathsf{of}(s), \mathcal{U}\_\mathsf{of}(s)).$$

The function $\mathcal{K}\_\mathsf{of}(s)$ reconstructs each key's version list from the corresponding server's state and extends it with the effects of all ongoing transactions' *client-committed* operations, i.e., those operations where the client has already committed in the two-phase commit protocol, but the corresponding server has not yet followed to their committed state. In particular, the client-committed reads are added to the appropriate reader sets and the writes are appended as new versions to the respective key's version list. This part of r is largely determined by the need to satisfy update correspondence, i.e., that abstract and concrete state updates commute with r.

The function $\mathcal{U}\_\mathsf{of}(s)$ reconstructs the abstract configuration's client views from the protocol's client-side configurations. This part of r is mostly constrained by the need to satisfy the view-related guards of the abstract commit event (wellformedness, canCommit, vShift, consWithSnapshot) when refining the protocol's committing events, and, to a lesser extent, by update correspondence.

To help define r, we often add *history variables* to the protocol configurations. These record additional information, such as the order of client commits for each key to reconstruct the abstract version lists, but they are not needed for the protocol execution itself. In particular, guards must not depend on them.

To establish the refinement's proof obligations, we must prove different protocol invariants, many of which are recurring and thus reusable for other protocols. The most important ones concern:

Freshness of transaction IDs The clients' current transaction ID is fresh, i.e., does not occur in the KVS until the commit.

Past and future transactions stating that the respective client and servers are in particular starting or end states, and

Views These invariants include view wellformedness, view closedness (for canCommit), and session guarantees (for vShift).

Many of these invariants directly imply related guards needed in the refinement of the abstract commit event. Moreover, there are invariants that are related to particular protocol mechanisms such as locking (cf. Section 4) and timestamps (cf. Section 5).

3.3.2 Refinement is sometimes insufficient. For verifying protocols against the stronger isolation levels such as SSER, it suffices to establish a refinement between the protocol and the (appropriate instance of) the abstract model. This is, for instance, the case for S2PL (Section 4). However, for protocols relying on *timestamps* to isolate transactions, such as those commonly seen for *optimistic* concurrency control or for achieving TCCv isolation, including Eiger-PORT+ (Section 5), it might be impossible to directly establish a refinement. One reason is as follows. Timestamp-based protocols usually use the timestamps to define an order on versions and to identify "safe-to-read" versions (corresponding to their view). Hence, to ensure that clients always read the latest version in their view, the refinement mapping must reconstruct the version lists of the abstract KVS in the order of their commit timestamps. Otherwise, the proof of the abstract guard consWithSnapshot will fail (see Section 3.2). However, the *execution order* of commits and the *order of the associated commit timestamps* may not coincide. We call such commits *inverted commits.* Having inverted commits in an execution may require *inserting* a key's new version to its version list rather than *appending* it. Since the abstract model only ever appends new versions at the end of the version lists, the refinement proof alone would fail for executions with inverted commits.

3.3.3 Refinement in combination with reduction. To address the above problem, we introduce an extra proof step to reorder the inverted commits before the refinement. To this end, we define a modified protocol model $\widehat{\mathcal{M}}$ that restricts transaction commits to those that do not introduce any inverted commits and we decompose the proof of (2) into the following two steps:

$$\mathsf{reach}(\mathcal{M}) = \mathsf{reach}(\widehat{\mathcal{M}}), \tag{4}$$

$$r(\mathsf{reach}(\widehat{\mathcal{M}})) \subseteq \mathsf{reach}(\mathcal{I}(\mathbb{IL})). \tag{5}$$

We prove (5) by refinement, which works for the restricted model. To prove (4), we use a proof technique based on Lipton's reduction method [34] to successively reorder inverted commits in executions by commuting causally independent events, while preserving the executions' final state. VerIso provides various sound proof rules to support such proofs. As we will see, verifying Eiger-PORT+ requires this combination of refinement and reduction.

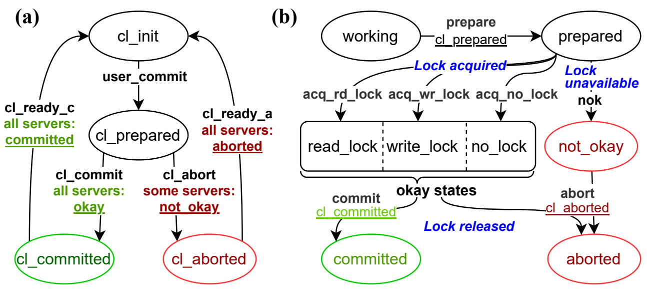

Figure 2: S2PL: state diagrams of (a) a client's `cl_state` and (b) a server's `svr_state` for a given transaction.

```
record cl_conf =            --- client configuration
  cl_state : state_cl
  cl_sn : sqn

record svr_conf =           --- server configuration
  svr_state : txid ⟶ state_svr
  svr_vl : v_list
  svr_fp : txid × {R,W} ⇀ value

record global_conf =        --- global configuration
  cls : cl_id ⟶ cl_conf
  svrs : key ⟶ svr_conf
```

Figure 3: S2PL client, server, and global configurations.

4 STRICT TWO-PHASE LOCKING

To illustrate VerIso's application, we start with a well-known distributed concurrency control protocol, namely Strict Two-Phase Locking (S2PL), which is commonly employed with strong isolation and reliability requirements. This protocol is usually combined with the Two-Phase Commit (2PC) protocol to achieve atomicity. We specify the combined S2PL+2PC protocol, still called S2PL for short, as an LTS and use VerIso to verify that it satisfies SSER.

4.1 Protocol Specification

The S2PL protocol is used in distributed systems with one transaction coordinator per client and several shard/replica servers that handle the clients' reads and writes to keys. In S2PL, each transaction acquires either exclusive write locks for writing to and reading from a key or shared read locks for just reading from a key. A transaction succeeds only if all the necessary locks for the involved keys are available. We assume that the failure to acquire any of the required locks results in aborting the whole transaction. In addition, 2PC guarantees a transaction's atomicity by ensuring that all servers involved in the transaction are prepared before committing. Figure 2 shows the control state diagrams of (a) a given client `cl`'s (coordinator) state and (b) a server's state for a given key `k` and transaction `t`. The state transitions are labeled with event names (in bold) and the transition guards (underlined). We now define the configurations and the events of our LTS model of S2PL, called TPL.

4.1.1 Configurations. We model the client, server, and global configurations as Isabelle/HOL records (Figure 3). The global configuration consists of mappings from client identifiers (`cl_id`) to client configurations and from keys to server configurations. Each client maintains a transaction state (`cl_state`) and a sequence number (`cl_sn`). As clients are sequential, there is only one transaction state per client, which may take one of four values as shown in Figure 2a. The sequence number keeps track of a client's current transaction.

Each server responsible for a given key `k`, called server `k` for short, keeps track of the ongoing transactions' states (`svr_state`), the key's version list (`svr_vl`), and the transactions' fingerprint on key `k` (`svr_fp`). Each transaction `t` is in one of the eight states depicted in Figure 2b and its fingerprint `svr_fp(k, t)` on `k` maps read and write operations to a value, if any.

Notation. Given a global configuration `s`, the client `cl`'s state, for instance, is accessed by `cl_state(cls(s,cl))`, where the functions `cls` of type `global_conf` ⟶ (`cl_id` ⟶ `cl_conf`) and `cl_state` of type `cl_conf` ⟶ `state_cl` are record field projections. For readability, we will write this as `cl_state(cl)` when `s` is clear from the context. We will also use `cl_state'(cl)` for the successor state `s'` of an event. Similar shorthands apply to all record fields.

4.1.2 Events. The client and server events change their respective transaction states as described in Figure 2. For brevity, we focus here on the server event `acq_write_lock` and the client event `cl_commit`. The latter plays a central role in the refinement proof.

Once a server `k` is in the prepared state for a given transaction `t`, it will try to lock its key `k` according to the operation performed on `k` and transition to the corresponding locked state (i.e., `read_lock`, `write_lock`, or `no_lock`). For simplicity of modeling, servers whose keys are not involved in `t` move to the state `no_lock`. Here is the definition of the server event `acq_write_lock`:

```
1 definition acq_wr_lock(k, v, w, t, s, s') ⟷
2   --- guards:
3   svr_state(k, t) = prepared ∧
4   cl_state(get_cl(t)) = cl_prepared ∧    --- client prepared
5   ∀t'. not_locked(svr_state(k, t')) ∧     --- no locks on k
6   w ∈ {⊥, last_ver_v(svr_vl k)} ∧
7   --- updates:
8   svr_state'(k) = write_lock ∧
9   svr_fp'(k, t) = [W ↦ v, R ↦ w]
```

This event acquires a write lock for transaction `t` to update the key `k` with the value `v` and possibly also for reading the key's latest value `w`, i.e., `w` is either ⊥ (no read) or `last_ver_v (svr_vl k)` (read) (Line 6). To execute the event, its guards require that (i) the server `k` is in the `prepared` state (Line 3), (ii) the transaction `t`'s client `get_cl(t)` is itself also in the `cl_prepared` state (Line 4), and (iii) no transaction has already acquired a lock on `k` (Line 5). Note that accessing the client's configuration corresponds to an abstract version of receiving a "prepared" message from the client. The event updates its state to `write_lock` (Line 8) and records its reads and writes in its key fingerprint variable `svr_fp` (Line 9).

Once all servers have reached a locked state by executing a lock-acquiring event, the next event is the client's `cl_commit` event:

```
definition cl_commit(cl, sn, u, F, s, s') ⟷
  --- guards:
  cl_state(cl) = cl_prepared ∧
  ∀k. is_locked(svr_state(k, get_txn(cl))) ∧
  sn = cl_sn(cl) ∧
  u = λk. full_view(𝒦_of_TPL(s, k)) ∧
  F = λk. svr_fp(k, get_txn(cl)) ∧
  --- updates:
  cl_state'(cl) = cl_committed
```

This event commits the client `cl`'s current transaction with sequence number `sn`, using the view `u` and the transaction fingerprint `F`. The latter two parameters are only used for the refinement proof. The guards (Lines 3–7) require that (i) the client is in its prepared state, (ii) all servers are in the locked state for the client's transaction `get_txn(cl)`, (iii) `sn` is the client's current sequence number, (iv) `u` corresponds to the *full view* of the current KVS' abstract version $\mathcal{K}$`_of_TPL(s)`, i.e., the view that includes all versions of all keys (see Section 4.2 for a more detailed description of $\mathcal{K}$`_of_TPL(s)`), and (v) `F` is the complete fingerprint for the client's transaction constructed from all servers' per-key fingerprints as set in the locking step. The client's new state is `cl_committed` (Line 9).

After the client's commit event, the transaction can no longer be aborted and all servers follow into their own commit state for the client's current transaction. In the following refinement proof, we show that this event refines the abstract model's commit event.

4.2 Refinement Proof

To verify that the S2PL protocol `TPL` guarantees strict serializability, we prove in Isabelle/HOL that the concrete protocol model, as just specified, refines the abstract model $\mathcal{I}(\mathsf{SSER})$.

Theorem 3. $\mathsf{TPL} \preccurlyeq_{r_{\mathsf{TPL}},\pi_{\mathsf{TPL}}} \mathcal{I}(\mathsf{SSER})$.

In the remainder of this section, we sketch our proof of this theorem.

4.2.1 Refinement mapping. We define the refinement mapping r_{TPL} on states, which consists of two components:

$\mathcal{K}$`_of_TPL`: This function reconstructs the global KVS' version lists from the version lists of all key servers and extends them with the effects of all ongoing transactions' *client-committed* operations (cf. Section 3.3).

$\mathcal{U}$`_of_TPL`: Since the client commit event is always using the *full view*, there is no need to store a concrete view in the client configurations. This function thus maps all concrete configurations to the (dummy) initial view $\lambda k.\,\{0\}$ of the abstract model.

The refinement mapping π_{TPL} on events maps `TPL`'s `cl_commit` event to the abstract commit event and all other events to the abstract skip event. To reconstruct the abstract commit event's parameters from the `cl_commit` event's parameters, we have added the former, namely, the intermediate view `u`, the fingerprint `F`, and the sequence number `sn` to `cl_commit`'s event parameters. Note that no event of `TPL` refines the abstract view extension event.

4.2.2 Proof of refinement (sketch). The main part of the proof consists of showing that for any concrete transition $s \xrightarrow{e} s'$ there is a transition $r_{\mathsf{TPL}}(s) \xrightarrow{\pi_{\mathsf{TPL}}(e)} r_{\mathsf{TPL}}(s')$ in $\mathcal{I}(\mathsf{SSER})$. For all events other than `cl_commit`, we must show that $r(s) = r(s')$, which corresponds to an abstract skip. This is the easy part. For the event `cl_commit(sn, u, F)`, we divide the proof that it refines the abstract commit event with the same parameters (cf. Section 3.2) into guard strengthening, i.e., that the concrete event's guards imply the abstract guards, and update correspondence (cf. Section 2.3). Let $\mathcal{K} = \mathcal{K}$`_of_TPL`$(s)$ and $\mathcal{K}' = \mathcal{K}$`_of_TPL`$(s')$.

For guard strengthening, we focus on two guards here: (i) The guard `canCommit(`$\mathcal{K}$`,u,F,`R_{SSER}`)` requires that the view `u` is *closed* under the relation $R_{\mathsf{SSER}} = \mathsf{WW}_{\mathcal{K}}^{-1}$, which means that all versions must be visible. VerIso provides a general, reusable lemma showing that this is the case for full views; and (ii) the guard `consWithSnapshot(`$\mathcal{K}$`,u,F)` requires that the values read by the transactions as indicated by the fingerprint `F` equal the latest values visible in the (full) view `u` of $\mathcal{K}$. This is the most interesting case. Our S2PL protocol model `TPL` satisfies this condition since the strict locking discipline ensures that, for each key, at most one transaction can write a new version to the KVS at a time. In particular, there cannot be any concurrent client-committed, but not yet server-committed writes to a key that the client's transaction reads. Proving this guard requires several invariants about locks and their relation to the fingerprint `F`, discussed below.

For update correspondence, we must establish that $\mathcal{K}'$ is equal to `UpdateKV(`$\mathcal{K}$`,Tn(cl,sn),u,F)`, which requires a series of lemmas about state updates. These in turn rely on VerIso's library lemmas about abstract state updates.

4.2.3 Required invariants and lemmas. The refinement proof uses numerous invariants and lemmas that describe the system's behaviour. First, it uses the generic invariants from Section 3.3.1, except view closedness, which trivially holds for the full views used in S2PL. Additionally, we need the following locking-related invariants for S2PL. These can be adapted to other lock-based protocols.

Lock invariants These invariants capture the multiple-reader, single-writer semantics of read and write locks.

Fingerprint invariants Three invariants relate each of the locking scenarios (read, write, or no lock) to the expected key fingerprint's content on key servers (read, write, or no operation).

5 EIGER-PORT+

We formally model a new, complex, distributed transaction protocol, called Eiger-PORT+, and verify its TCCv isolation guarantee with VerIso. This protocol leverages *timestamp-based optimistic concurrency control* rather than locking as in S2PL. We also demonstrate its superior performance over the state-of-the-art.

5.1 Eiger-PORT+ in a Nutshell

Causally consistent transactions have attracted the attention of both academia and industry in recent years. Eiger [40] is among the first distributed databases providing TCCv. Eiger-PORT [43] improves Eiger's overall performance by optimizing its read-only transactions while sacrificing its support for transactional writes *and* data convergence (thus rendering diverging client views of concurrent conflicting writes as in TCC).

We show that this sacrifice is unnecessary and design Eiger-PORT+ based on Eiger-PORT. Eiger-PORT+ provides both causally consistent read- and write-only transactions with the data convergence guarantee.[1] The key idea of achieving convergent client views is to share with clients the *already established total order* of versions on a server (by uniquely assigning timestamps across versions); this differs from Eiger-PORT, which constructs individual, possibly different, orders per client. Consequently, Eiger-PORT+ improves performance by eliminating the overhead of scanning long-standing, stale versions to locate the correct version to return.

[1]Eiger-PORT+'s pseudo code, together with its description, is given in Appendix A

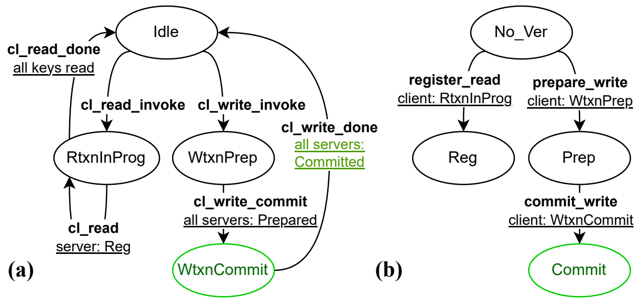

Figure 4: Eiger-PORT+: state diagrams of (a) a client's `cl_state` and (b) a server's `svr_state` for a given transaction.

Eiger-PORT's authors conjecture that TCC is the upper bound of the achievable isolation levels for performance-optimal read-only transactions in the presence of transactional writes [43]. Our new protocol and its TCCv correctness proof refute this conjecture. Moreover, we show that Eiger-PORT+ outperforms both Eiger-PORT (providing TCC) and its precursor Eiger (providing TCCv).

5.2 Differences to the S2PL Protocol

We summarize the main differences and challenges compared to S2PL (Section 4) with respect to formal design and verification.

5.2.1 Protocol design and modeling. Eiger-PORT+'s write transactions follow a variant of the 2PC protocol that always commits [40]. Hence, in contrast to the S2PL protocol, there are no abort states here (cf. Figure 4). Instead of using locks and always reading a key's latest version, the Eiger-PORT+ protocol uses timestamps based on Lamport clocks and reads the latest, possibly stale version below a certain timestamp, called the client's *global safe time* (or the client's latest own write, if newer). This required more involved definitions and additional invariants supporting the correctness proof.

Rather than storing fingerprints and version lists separately, our Eiger-PORT+ model records relevant information about a transaction directly in the state constructor's parameters. For example, the server state `Commit(cts, sclk, slst, v, rs)` includes the commit timestamp `cts`, the value `v`, the reader set `rs`, and other timestamps.

5.2.2 Protocol correctness proof. The reconstruction of the abstract configuration from the protocol configuration is based on the transactions' states and on a history variable (`cts_order`), which records the timestamp-based commit order of transactions for each key and, together with the transactions' states, roughly corresponds to S2PL's version lists. Since Eiger-PORT+ exhibits inverted commits (which is allowed by TCCv), we may have to *insert* new transactions into the `cts_order`'s lists. This is in contrast to S2PL's version lists, to which new versions are only *appended*. Therefore, Eiger-PORT+'s correctness proof requires using refinement in combination with reduction, for which VerIso provides various proof rules.

For the refinement proof, most proof obligations (cf. Section 3.2) become harder to prove than for S2PL. For example, (i) the guard `canCommit` for TCCv requires the views to be closed under causal dependencies, rather than clients having simply the complete view of all versions; we prove an invariant stating that the views remain closed, which in turn requires numerous auxiliary invariants about

```
record cl_conf =                         --- client configuration
  cl_state : txn_state
  cl_sn : sqn
  cl_clock : ts
  gst : ts
  lst_map : key ⟶ ts

record 'v svr_conf =                     --- server configuration
  svr_state : txid ⟶ ver_state
  svr_clock : ts
  lst : ts

record 'v global_conf =                  --- global configuration
  cls : cl_id ⟶ cl_conf
  svrs : key ⟶ svr_conf
  cts_order : key ⟶ list(txid)  --- history variable
```

Figure 5: Eiger-PORT+ configurations.

timestamps; (ii) the `consWithSnapshot` condition's proof is complicated by the more complex definition of reading at a timestamp; this requires proving lemmas about the reading functions; and (iii) the `viewShift` condition is no longer trivial, expressing the monotonic growth of view and the *read-your-writes* session guarantee.

5.3 Formalizing Eiger-PORT+

Figure 4 shows the client and server's transaction states and summarizes the events that transition between these states and their guards. As with S2PL, we model the client, server, and global configurations as records in Isabelle/HOL (Figure 5); the global configuration contains the client (`cls`) and server (`svrs`) configurations, and history variables, which we discuss in Section 5.4.1.

5.3.1 Timestamps. Eiger-PORT+ timestamps committed writes based on Lamport clocks [31]. These clocks respectively correspond to the client and server variables `cl_clock` and `svr_clock`. In order to obtain globally unique and totally ordered commit timestamps, the clock reading is paired with the transactions' client IDs and these pairs are ordered lexicographically.

Each server maintains a *local safe time* `lst` that corresponds to the minimum of the pending transactions' timestamps or, if there is none, the maximum committed timestamp for that server. Each client maintains a variable `lst_map`, which maps server identifiers to their `lst` value, and a *global safe time* `gst`. The latter is updated to the minimum timestamp in `lst_map` when a read-only transaction starts, and acts as the *stable frontier* for that client (all transactions with earlier timestamps are guaranteed to be committed on the server). Each read sent to a server includes `gst` as a read timestamp.

5.3.2 Read-only transactions. The event sequence for read transactions is as follows (see Figure 4). A client's `cl_read_invoke` event starts a read-only transaction and transitions from the `Idle` to the `RtxnInProg(clk, keys, ∅)` state, where `clk` is the client's current clock reading, `keys` is the (finite and non-empty) set of keys to be read, and `∅` is the empty key-value mapping, where the subsequently read values will be recorded. As mentioned, this event also updates the client's global safe time, `gst`, to the minimum of the servers' local safe times stored in `lst_map`. As a result, more up-to-date versions of certain keys may become visible to the client.

Once a client has invoked a read, the involved servers (in `keys`) follow with a `register_read` event, where they access the client's

```
1  definition cl_write_commit(cl, kv_map, cts, sn, u, clk, s, s')
       ⟷
2    --- guards:
3    cl_state(cl) = WtxnPrep(kv_map) ∧
4    (∀k ∈ dom(kv_map). is_prepared(s, k, cl, kv_map) ∧
5    cts = Max {get_prepared_ts(svr_state(k, Tn(sn,cl))) |
6              k ∈ dom(kv_map)} ∧
7    sn = cl_sn(cl) ∧
8    u = 𝒰_of_EPP(s, cl) ∧
9    clk = cts + 1 ∧
10   --- updates:
11   cl_state'(cl) = WtxnCommit(cts, kv_map) ∧
12   cl_clock'(cl) = clk ∧
13   cts_order' = extend_cts_order(s, Tn(sn, cl), cts, kv_map)
```

Figure 6: Eiger-PORT+'s client commit event.

gst and determine the latest transaction with a commit timestamp cts ≤ gst, or the latest own write newer than the gst (if any). This transaction is recorded in the read version's readerset rs, along with the current lst and updated server clock reading. This information is then accessed in the client's subsequent cl_read event, which reads the version's value and updates its own clock and lst_map. Finally, when the client has read all requested values, i.e., its state RtxnProg(clk, keys, kv_map) is such that dom(kv_map) = keys, the event cl_read_done brings it back to the Idle state.

5.3.3 Write-only transactions. Write transactions are initiated by the client's cl_write_invoke event, in which the client transitions from Idle to the state WtxnPrep(kv_map). The key-value map kv_map describes the keys and associated values to be written and corresponds to the transaction's (write-only) fingerprint. Once all servers have followed into their prepared state, the client can execute the cl_write_commit event, which is defined in Figure 6.

This event has seven parameters: the client ID cl, the key-value map kv_map, the commit timestamp cts, which is the maximum of the involved server's prepared timestamps for the current transaction Tn(sn, cl) (Lines 5-6), the client's current sequence number sn (line 7), the abstract view u (Line 8, see Section 5.4), and the updated Lamport clock clk (Line 9). As cts > cl_clock always holds here, there is no need to take their maximum to determine clk. The guards at Lines 3 and 4 require that the client is in the prepared state and that all involved servers have followed into their own prepared state. The client's state is updated to WtxnCommit(cts, kv_map) (Line 11) and the clock is updated (Line 12). We will discuss the history variable update at Line 13 in Section 5.4.

After the client's commit event, the involved servers follow and commit the transaction on their side using the commit_write events. When all servers have committed, the client executes the cl_write_done event, returning it to the idle state.

5.4 Refinement Mapping

The refinement mapping π_{EPP} on events maps the cl_read_done and cl_write_commit events to the abstract commit event, the event cl_read_invoke to the abstract view extension event, and all other events to skip. The refinement mapping on protocol configurations s is defined by two components:

$$r_{\mathsf{EPP}}(\mathtt{s}) = (\mathcal{K}\_\mathtt{of\_EPP}(\mathtt{s}),\ \mathcal{U}\_\mathtt{of\_EPP}(\mathtt{s})).$$

We now describe the protocol model's history variables and the functions 𝒦_of_EPP and 𝒰_of_EPP, which reconstruct the abstract model's KVS and client views using these history variables.

5.4.1 History variables. The global configuration includes three history variables, which help reconstruct abstract model configurations from protocol model configurations. Here, we only discuss the most important one. The variable cts_order maps each key to a list of client-committed write-only transactions, *ordered by their commit timestamps.* The client commit event extends the cts_order by *inserting* the committed transaction's ID at the position corresponding to its commit timestamp into the transaction ID list of each key written by the transaction (Line 13 in Figure 6).

5.4.2 Abstract KVS. The function 𝒦_of_EPP reconstructs the abstract KVS from the cts_order history variable by mapping each key's list of client-committed transactions to an abstract version list. For a given key k and transaction t, we extract each version's value and readerset from server k's (prepared or committed) state for t. Since the abstract model always reads the latest versions in a client's view, as expressed by the guard consWithSnapshot of the abstract commit event (see Section 3.2), the cts_order must be sorted by commit timestamp (cf. also Section 3.3).

5.4.3 Abstract views. We define the function 𝒰_of_EPP, reconstructing the abstract views from Eiger-PORT+'s model configurations in two steps. We first construct a function get_view, where get_view(s, cl, k) denotes the set of client-committed transactions t, whose commit timestamp is less than or equal to the client's gst, i.e., wtxn_cts(t) ≤ gst(cl) or that are the client cl's own transactions. Second, we use cts_order to map the transactions IDs in the range of get_view to their positions in the cts_order, which correspond to indices into the abstract version lists.

5.5 Correctness: Eiger-PORT+ satisfies TCCv

To prove that Eiger-PORT+ guarantees TCCv, we instantiate the abstract model with $R_{\mathsf{TCCv}} = \mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}$ and the $\mathsf{vShift}_{\mathsf{TCCv}}$ predicate, which requires monotonic reads and read-your-writes.

We can now state our main result of this section.

Theorem 4 (Correctness of EP+). *The Eiger-PORT+ model* EPP *satisfies* TCCv, *i.e.,* $r_{\mathsf{EPP}}(\mathsf{reach}(\mathsf{EPP})) \subseteq \mathsf{reach}(\mathcal{I}(\mathsf{TCCv}))$.

As Eiger-PORT+ exhibits inverted commits, our proof combines reduction and refinement, as outlined in Sections 3.1 and 3.3. We sketch both parts of the proof, stated as Lemmas 5 and 6 below, followed by describing the invariants used in these proofs.

5.5.1 Restricted model and reduction proof. We define the restricted model $\widehat{\mathsf{EPP}}$ by adding a guard to the event cl_write_commit, which requires that the unique commit timestamp (cts, cl) of the client's transaction is greater than any commit timestamp of a transaction in the cts_order. This ensures that the client commit only appends, but does not insert, the new transaction into the cts_order. For this model, we prove the following lemma.

Lemma 5. $\mathsf{reach}(\mathsf{EPP}) = \mathsf{reach}(\widehat{\mathsf{EPP}})$.

Proof (sketch). By construction of $\widehat{\mathsf{EPP}}$, the right-to-left inclusion is easily shown by a refinement. For the left-to-right inclusion, consider any execution e of EPP ending in some state s. We prove by reduction that we can reorder all inverted commits in e of EPP, while preserving its final state s. We first show that the relevant events of two transactions with inverted commit timestamps are pairwise causally independent. We then prove that adjacent causally independent events in e can be commuted. Using a measure function on executions, we to show that this process terminates in an execution $\widehat{e}$ without inverted commits ending in state s, thus an execution of $\widehat{\mathsf{EPP}}$. Hence, any state reachable in EPP is also reachable in $\widehat{\mathsf{EPP}}$. □

5.5.2 Refinement proof. Next, we establish a refinement between the restricted model $\widehat{\mathsf{EPP}}$ and the abstract model $\mathcal{I}(\mathsf{TCCv})$ instantiated to TCCv, using the refinement mapping defined in Section 5.4.

LEMMA 6. $r_{\mathsf{EPP}}(\mathsf{reach}(\widehat{\mathsf{EPP}})) \subseteq \mathsf{reach}(\mathcal{I}(\mathsf{TCCv}))$.

PROOF (SKETCH). We show guard strengthening and update correspondence for every event of $\widehat{\mathsf{EPP}}$. This is easy for most events, which refine skip. The interesting cases are the read invoke event, which refines the abstract view extension event, and the client commit and read done events, refining the abstract commit event.

Here, we focus on the client commit event. The update correspondence proof relies on the absence of inverted commits in $\widehat{\mathsf{EPP}}$ and thus versions being appended to KVS version lists by client commits. For the guard strengthening, we must show that all guards of the abstract commit event (cf. Section 3.2) are implied by the concrete guards. We discuss the most interesting ones. View atomicity (part of view wellformedness) holds by construction, since all versions of a transaction have the same `cts` and the abstracted view includes all transactions with a `cts` below the client's `gst` (and also its own transactions). By a similar argument, we prove that consWithSnapshot holds, i.e., a client reads the latest version in its view. To show that canCommit holds, we prove an invariant stating that the clients' views remain closed with respect to R_{TCCv}. This proof in turn requires several invariants about timestamps. □

5.5.3 Invariants and lemmas. Our proofs rely on numerous invariants and lemmas. In addition to the generic ones from Section 3.3.1, we need the following timestamp-related lemmas and invariants. These can easily be adapted to other timestamp-based protocols.

Timestamps This category includes monotonicity lemmas showing the monotonic increase of timestamp variables and the following invariant for all clients `cl` and servers `k`:
`gst(cl) < lst_map(cl, k) < lst(k) < svr_clock(k)`.

Client commit order These invariants express that the `cts_order` history variable is sorted by commit timestamps, has distinct elements, and contains only client-committed transactions.

5.6 Performance Evaluation

5.6.1 Workloads and setup. For a fair comparison, we implement Eiger-PORT+ in the same codebase of Eiger and Eiger-PORT. We also use the same YCSB-like dynamic workload generator. Its default parameters are: 32 threads per client, 1 million keys, 128-byte values, 5 columns per key, 5 keys per operation, 90% read proportion, and the Zipfian key-access distribution with the skew factor of 0.8.

Our experiments run on a CloudLab [19] cluster of machines, each with 2.4GHz Quad-Core Xeon CPU, 12GB RAM, and 1Gbps network interface. We employ the same primary-backup replication

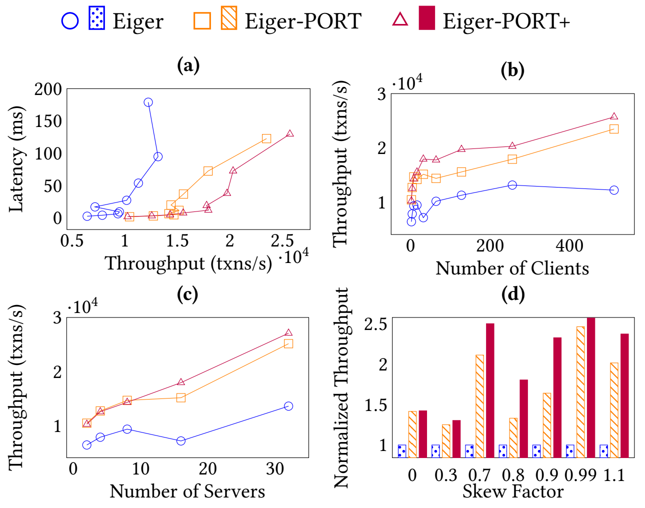

Figure 7: Performance comparison.

as in Eiger and Eiger-PORT, with two logical data centers co-located in the cluster. By default, each data center uses eight servers to partition the database and eight client machines to load the servers. We plot each data point using the average over five 60-second trials.

5.6.2 Comparison results. Overall, Eiger-PORT+ is highly performant and superior to both competing algorithms (Figure 7a). In particular, compared to the performance-optimal [43] transaction algorithm Eiger-PORT providing TCC, Eiger-PORT+ exhibits *higher* throughput with a *stronger* isolation guarantee of convergence. Eiger-PORT+ also scales well with an increasing number of clients (Figure 7b) or servers (Figure 7c), with up to 1.8x (resp. 2.5x) throughput improvement over Eiger-PORT (resp. Eiger). In addition, despite varying skews, the throughput of Eiger-PORT+ consistently surpasses that of its competitors (Figure 7d). This improvement becomes more pronounced with highly skewed workloads (or larger skew factors). This is because higher skewness results in increased concurrency, which would trigger additional server-side computation in Eiger-PORT and more rounds of communication in Eiger.

6 FINDING ISOLATION BUGS USING VERISO

We have shown how VERISO can be employed to verify database designs. In this section, we demonstrate how VERISO can also help uncover different kinds of isolation bugs in protocols. Our approach to finding these bugs is as follows.

We set up a refinement proof of the claimed isolation guarantee in VERISO as usual for protocol verification. Following the technique in Section 3.3, we define a refinement mapping and attempt to prove all proof obligations. If a proof obligation fails, we check whether commuting events could solve the problem. If so, we try to prove by reduction that commuting the relevant events preserves the set of reachable states. In the case of SSER, we must not commute events of non-overlapping transactions to preserve their real-time order. If the reduction proof fails or reduction cannot solve the problem, we construct a counterexample by investigating the failed proof state.

6.1 Case Study: TAPIR Protocol

We use the TAPIR [60] protocol, which claims to provide SSER, to demonstrate our approach. TAPIR is a distributed transaction protocol, consisting of 2PC and optimistic concurrency control (OCC), which claims to offer *strictly serializable* read-write transactions. Before committing a transaction, TAPIR performs a *timestamp-based* OCC validation, checking for any concurrent conflicting transactions. Moreover, by building it on an *inconsistent replication* protocol, the entire transaction system can offer fault tolerance while eliminating the coordination redundancy found in conventional replicated databases like Spanner.

Our modeling of TAPIR is similar to that for S2PL, except that (i) TAPIR performs an OCC check instead of locking when preparing a read or write on a server, and (ii) the client's state contains a local time for generating timestamps. Using VerIso, we discover that TAPIR violates *atomic visibility*, in addition to the previously found issue with real-time ordering [42]. The authors of [42] conjecture that TAPIR still provides SER; however, violating atomic visibility indicates that it does not even satisfy RA, a much weaker isolation level. Neither the TAPIR authors' manual correctness proofs (including the SSER guarantee) nor their additional model checking analysis using TLA+ caught these bugs. Due to a lack of experimental documentation, it is unclear why their model checking analysis missed all these bugs. This underlines the need for our *rigorous* verification approach, which covers *all* system behaviors.

6.2 Finding TAPIR Isolation Bugs using VerIso

We set up a SSER refinement proof for TAPIR in VerIso. For the refinement mapping, we use the same technique as in Eiger-PORT+ to reconstruct the abstract KVS by retaining a commit order. However, there is now only one client commit event simulating the abstract atomic commit. As TAPIR claims to provide SSER, the view must stay closed under $R_{\mathsf{SSER}} = \mathsf{WW}_{\mathcal{K}}^{-1}$, i.e., each client's view is a full view, containing all client-committed versions.

With the lemmas for full views, we prove the view wellformedness and closure (wf and canCommit). However, we are unable to prove the `consWithSnapshot` guard strengthening due to the two problems below, leading us to construct our two counterexamples.

6.2.1 Non-atomic Visibility. This problem arises from the fact that TAPIR allows reading from a key when there is already a prepared and client-committed write on the same key. A reading event on a server reads the latest version committed on that server, while the client's view contains all *client-committed* versions. Therefore, it is possible for a read not to read the latest version in the view when the version is only client-committed but not yet server-committed. The following shows a TAPIR execution illustrating this problem.

Example 7. As shown in Figure 8a, the transaction tx_1 with the timestamp $t_1 = 8$ has already prepared its writes to the keys A and B and committed on A; meanwhile, the transaction tx_2 attempts to prepare a read on the same keys with the timestamp $t_2 = 5$ and succeeds (based on TAPIR's OCC validation check [60][2]). The read on A returns the recently written version W_1^A, while the read on

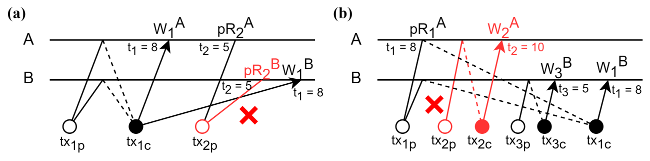

Figure 8: Counterexamples for SSER claimed by TAPIR: (a) an atomic visibility violation and (b) timestamp inversion. p and c refer to clients' *prepare* and *commit* events, respectively.

B fetches the initial version. This demonstrates a *fractured read* anomaly (Section 2.1): tx_1's writes are partially observed by tx_2.

Reduction cannot help here as a single server's events do not commute. Hence, we can neither move the pR_2^B prepare event to the right nor the pR_2^A prepare event to the left to avoid the problem.

6.2.2 Timestamp Inversion. TAPIR was recently found to exhibit timestamp inversion [42]. This problem happens when a transaction overlaps with a set of non-overlapping transactions whose real-time order differs from their timestamp order, ultimately leading to a violation of SSER.[3] We rediscover this bug using VerIso.

Since the client has a full view, proving the `consWithSnapshot` guard requires that the version read by the transaction is the latest in the KVS. However, TAPIR allows new versions to be committed on servers with prepared reads if the versions have a higher timestamp than all prepared reads' timestamps. Hence, a prepared read on a server might no longer correspond to the latest version in the KVS when committing, which makes the `consWithSnapshot` guard fail.

In principle, postponing the events of a transaction that has written a new version on the server could solve this problem. We thus attempt at a reduction proof, restricted to commuting events of overlapping transactions (to preserve SSER's real-time order). However, a non-overlapping transaction can block the necessary commute of events, which means that reduction cannot fix the guard failure. This is illustrated in the following counterexample. Note that the timestamps pass TAPIR's OCC validation check.

Example 8. Figure 8b depicts a TAPIR execution leading to a timestamp inversion. The transaction tx_1 with the timestamp $t_1 = 8$ has already prepared a read on the key A when a new transaction tx_2 with a higher timestamp $t_2 = 10$ succeeds in committing a new version on A. At this point, the `consWithSnapshot` guard does not hold unless we can move the events of tx_2 after the first transaction's client commit (tx_{1c}). However, as the transaction tx_3 does not overlap with tx_2, moving tx_2's events to the right would require moving tx_3's events as well to preserve these transactions' real-time order. This is impossible because $t_1 > t_3$ and hence commuting the client commit events of these two transactions would introduce inverted commits. So the reduction proof fails as well.

Our step-by-step approach facilitates spotting isolation bugs by breaking down the protocol verification proof into smaller, more manageable parts, i.e., proof obligations, each corresponding to different protocol aspects. Overall, these counterexamples illustrate

[2]The OCC validation check `tapir_occ_check` is different in the conference paper [59], for which we also provide an atomic visibility counterexample in Appendix B.

[3]Unlike timestamp inversion, *inverted commits* introduced in Section 3.3 concern *all* transactions, not just non-overlapping ones, and are *not* considered an isolation bug.

how our technique can be used to discover both known and new isolation bugs of different kinds. Note that these counterexamples are also valid when TAPIR is layered on top of the replication protocol.

7 RELATED WORK

Along with the advances in designing reliable, performant database transactions, considerable efforts have been dedicated to formalizing their isolation guarantees. We classify them into two categories.

Declarative Specifications. *Dependency graphs* were first introduced by Bernstein et al. [8] for characterizing serializability in early 1980s, which were later adapted to weaker isolation levels such as snapshot isolation and read committed by Adya [1].

Based on *abstract executions*, Cerone et al. [12] propose an axiomatic framework to declaratively define isolation levels at least as strong as read atomicity, with the dual notions of visibility (what transactions can observe) and arbitration (the order of installed versions/values). To ease the validations of isolation guarantees over database execution histories, Biswas and Enea [9] present an alternative axiomatic framework for characterizing isolation criteria via the write-read relation which, in contrast to the visibility relation [12], can be extracted straightforwardly from histories.

Recent years have seen the development of effective isolation checkers based on some of these declarative specifications [9, 22, 25, 30, 56, 61]. These include Biswas and Enea's dbcop, based on their axiomatic framework [9], and Elle [30], based on Adya's dependency graph [1], for checking multiple isolation levels, as well as Cobra [56] and PolySI [25] tailored to serializability and snapshot isolation, respectively. All these checkers verify whether single database execution histories satisfy the isolation level in question. Hence, they can only show the presence of errors, not their absence, in database implementations. Given that many isolation bugs have stemmed from design-level defects in carefully designed and heavily tested production databases, full verification of database designs is highly desirable, preferably at an early design stage. Our mechanized verification framework fulfills this requirement.

Operational Specifications. Crooks et al. [15] introduce a state-based formalization of isolation guarantees, relying on complete transaction traces; VerIso only uses the current database state and poses restrictions on the transactions visible to clients. Liu et al. [38] develop a formal framework in the Maude language [14] for specifying and model checking data consistency properties with respect to a database execution history of transactions. Lesani et al. [32] propose an operational framework for the verification of causally-consistent key-value stores.

We base our framework VerIso on the operational characterization of isolation guarantees in [58]. VerIso distinguishes itself from the aforementioned related work by supporting *invariant-based verification*, which can be mechanized using theorem provers like Isabelle/HOL. Most of these related works [1, 8, 12, 15, 58] only provide a theoretical formalization of isolation criteria with no tool support for formal analysis. Similar to VerIso, the framework in [32] is formalized in the Coq proof assistant, supporting mechanized theorem proving. However, it is only compatible with non-transactional key-value stores and causal consistency. The framework in [38] supports transactional databases and is accompanied by an explicit-state model checker. However, it offers weaker correctness guarantees, limited to a small number of processes and transactions. In contrast, VerIso supports rigorous verification, regardless of how many processes and transactions are involved.

Table 1: Statistics of the Isabelle/HOL formalization.

	Tot LoC	Model LoC	Proof LoC	#lemmas	#inv.
VerIso framew.	3093	–	–	275	11
S2PL+2PC	1890	285	1605	70	12
Eiger-PORT+	10318	746	9572	490	108
TAPIR	951	386	565	14	12
Total	16252	4510	11742	849	143

8 DISCUSSION AND CONCLUDING REMARK

We have presented VerIso, the first systematic, mechanized framework for formally verifying the correctness of database transaction protocols with respect to a wide range of isolation levels. We have demonstrated its effectiveness through three case studies of different kinds. Table 1 shows the statistics of our formalization effort.

Given repeated occurrences of isolation bugs with respect to database protocol designs, our work can pave the way to designing reliable and high-performance transaction systems. The following characteristics of our framework are relevant here.

Modeling and Proving Infrastructure. VerIso provides both methodological guidance and concrete structural support for verifying protocols. We have outlined a modeling style for protocols (Section 3), exemplified in our three case studies. Moreover, we have identified several categories of invariants reusable or adaptable to other protocol verification efforts. We also offer a concrete verification infrastructure, which consists of several elements: the abstract model and its eight instantiations to express the desired isolation levels, a sizable collection of supporting lemmas (cf. the first row of Table 1) configured to increase proof automation, and a set of refinement and invariant proof rules embedded in Isabelle/HOL.

Strong Correctness Guarantees. Our protocol correctness proofs provide strong guarantees. In particular, they hold for an arbitrary number of clients and servers. This guarantee cannot be provided by other methods such as model checking and testing. Moreover, Isabelle/HOL is a foundational framework with a very small, well-tested logical kernel, which provides strong soundness guarantees.

Generality and Extensibility. VerIso covers a wide spectrum of isolation levels ranging from RA to SSER, rather than being specialized for a particular level. Given its parametric nature, VerIso can be instantiated with other guarantees (e.g., *non-monotonic snapshot isolation* [4] and *regular sequential serializability* [23]), provided they are at least as strong as RA. Therefore, VerIso can be applied to a large variety of existing and new protocols.

Our long-term goal is to make VerIso more generic to cover an even wider range of database transaction protocols and isolation guarantees, including weaker ones like RC that do not offer atomic visibility. Moreover, it would be desirable to extend the verification guarantees from the design to implementation, where the approach in [54] may offer an interesting avenue for future work.

REFERENCES

[1] Atul Adya. 1999. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions.* Ph. D. Dissertation. Massachusetts Institute

of Technology, Department of Electrical Engineering and Computer Science.
[2] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal Memory: Definitions, Implementation, and Programming. *Distributed Comput.* 9, 1 (1995), 37–49.
[3] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno M. Preguiça, and Marc Shapiro. 2016. Cure: Strong Semantics Meets High Availability and Low Latency. In *ICDCS 2016*. IEEE Computer Society, 405–414.
[4] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. 2013. Non-monotonic Snapshot Isolation: Scalable and Strong Consistency for Geo-replicated Transactional Systems. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*. 163–172.
[5] Azure. 2022. azure-cosmos-tla. https://github.com/Azure/azure-cosmos-tla.
[6] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2016. Scalable atomic visibility with RAMP transactions. *ACM Transactions on Database Systems (TODS)* 41, 3 (2016), 1–45.
[7] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record* 24, 2 (1995), 1–10.
[8] Philip A Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)* 13, 2 (1981), 185–221.
[9] Ranadeep Biswas and Constantin Enea. 2019. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 165:1–165:28.
[10] Manuel Bravo, Alexey Gotsman, Borja de Régil, and Hengfeng Wei. 2021. UniStore: A fault-tolerant marriage of causal and strong consistency. In *USENIX ATC 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 923–937.
[11] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. 2015. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs, Vol. 37)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 568–590.
[12] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *CONCUR'15 (LIPIcs, Vol. 42)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 58–71.
[13] Audrey Cheng, Xiao Shi, Lu Pan, Anthony Simpson, Neil Wheaton, Shilpa Lawande, Nathan Bronson, Peter Bailis, Natacha Crooks, and Ion Stoica. 2021. RAMP-TAO: Layering Atomic Transactions on Facebook's Online TAO Data Store. *Proc. VLDB Endow.* 14, 12 (2021), 3014–3027.
[14] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. 2007. *All about Maude - a High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. Springer-Verlag, Berlin, Heidelberg.
[15] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC'17)*. ACM, 73–82.
[16] A. Jesse Jiryu Davis, Max Hirschhorn, and Judah Schvimer. 2020. Extreme Modelling in Practice. *Proc. VLDB Endow.* 13, 9 (may 2020), 1346–1358. https://doi.org/10.14778/3397230.3397233
[17] Diego Didona, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. 2018. Causal Consistency and Latency Optimality: Friend or Foe? *Proc. VLDB Endow.* 11, 11 (2018), 1618–1632.
[18] Wensheng Dou, Ziyu Cui, Qianwang Dai, Jiansen Song, Dong Wang, Yu Gao, Wei Wang, Jun Wei, Lei Chen, Hanmo Wang, Hua Zhong, and Tao Huang. 2023. Detecting Isolation Bugs via Transaction Oracle Construction. In *ICSE '23*. IEEE Press, 1123–1135.
[19] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *USENIX ATC'19*. 1–14.
[20] ElectricSQL. 2024. https://electric-sql.com/.
[21] Shabnam Ghasemirad, Si Liu, Luca Multazzu, Christoph Sprenger, and David Basin. 2024. *VerIso: Verifiable Isolation Guarantees for Database Transactions*. Technical Report. ETH Zurich. https://github.com/ShabnamRad/VerIso.
[22] Long Gu, Si Liu, Tiancheng Xing, Hengfeng Wei, Yuxing Chen, and David Basin. 2024. IsoVista: Black-box Checking Database Isolation Guarantees. *Proc. VLDB Endow.* 17, 12 (2024).
[23] Jeffrey Helt, Matthew Burke, Amit Levy, and Wyatt Lloyd. 2021. Regular Sequential Serializability and Regular Sequential Consistency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. ACM, 163–179.
[24] Diana Hsieh. 2017. CockroachDB beta passes Jepsen testing. https://www.cockroachlabs.com/blog/cockroachdb-beta-passes-jepsen-testing/.
[25] Kaile Huang, Si Liu, Zhenge Chen, Hengfeng Wei, David A. Basin, Haixiang Li, and Anqun Pan. 2023. Efficient Black-box Checking of Snapshot Isolation in Databases. *Proc. VLDB Endow.* 16, 6 (2023), 1264–1276.
[26] Jepsen. Accessed in May, 2024. Jepsen Analyses. https://jepsen.io/analyses.
[27] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. 2023. Detecting Transactional Bugs in Database Engines via Graph-Based Oracle Construction. In *OSDI'23*. USENIX Association.
[28] Kyle Kingsbury. 2018. Dgraph 1.0.2. https://jepsen.io/analyses/dgraph-1-0-2.
[29] Kyle Kingsbury. 2019. YugaByte DB 1.3.1. https://jepsen.io/analyses/yugabyte-db-1.3.1,polysi.
[30] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (2020), 268–280.
[31] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (jul 1978), 558–565.
[32] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: certified causally consistent distributed key-value stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 357–370. https://doi.org/10.1145/2837614.2837622
[33] Galera Cluster Library. 2023. Remove SNAPSHOT ISOLATION mention from FAQ page #349. https://github.com/codership/documentation/commit/cc8d6125f1767493eb61e2cc82f5a365ecee6e7a.
[34] Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (1975), 717–721.
[35] Si Liu. 2022. All in One: Design, Verification, and Implementation of SNOW-Optimal Read Atomic Transactions. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 43 (mar 2022).
[36] Si Liu, Luca Multazzu, Hengfeng Wei, and David A. Basin. 2024. NOC-NOC: Towards Performance-optimal Distributed Transactions. *Proc. ACM Manag. Data* 2, 1, Article 9 (mar 2024).
[37] Si Liu, Peter Csaba Ölveczky, Qi Wang, Indranil Gupta, and José Meseguer. 2019. Read atomic transactions with prevention of lost updates: ROLA and its formal analysis. *Formal Aspects Comput.* 31, 5 (2019), 503–540.
[38] Si Liu, Peter Csaba Ölveczky, Min Zhang, Qi Wang, and José Meseguer. 2019. Automatic Analysis of Consistency Properties of Distributed Transaction Systems in Maude. In *TACAS'19 (LNCS, Vol. 11428)*. Springer, 40–57.
[39] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP 2011*. ACM, 401–416.
[40] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *NSDI 2013*. USENIX Association, 313–328.
[41] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. 2016. The SNOW Theorem and Latency-Optimal Read-Only Transactions. In *OSDI 2016*. USENIX Association, 135–150.
[42] Haonan Lu, Shuai Mu, Siddhartha Sen, and Wyatt Lloyd. 2023. NCC: Natural Concurrency Control for Strictly Serializable Datastores by Avoiding the Timestamp-Inversion Pitfall. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, 305–323.
[43] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. 2020. Performance-Optimal Read-Only Transactions. In *OSDI 2020*. USENIX Association, 333–349.
[44] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 2017. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *NSDI 2017*. USENIX Association, 453–468.
[45] Microsoft. 2024. Azure CosmosDB DB. https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels.
[46] MongoDB. 2024. https://www.mongodb.com/.
[47] Neo4j. 2024. https://neo4j.com/.
[48] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon Web Services uses formal methods. *Commun. ACM* (2015). https://www.amazon.science/publications/how-amazon-web-services-uses-formal-methods
[49] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer. https://doi.org/10.1007/3-540-45949-9
[50] Christos H Papadimitriou. 1979. The serializability of concurrent database updates. *Journal of the ACM (JACM)* 26, 4 (1979), 631–653.
[51] Matthieu Perrin, Achour Mostefaoui, and Claude Jard. 2016. Causal Consistency: Beyond Memory. *SIGPLAN Not.* 51, 8, Article 26 (Feb. 2016), 12 pages.
[52] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 385–400.
[53] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. 2021. Optimistic Causal Consistency for Geo-Replicated Key-Value Stores. *IEEE Trans. Parallel Distributed Syst.* 32, 3 (2021), 527–542.
[54] Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix Wolf, Peter Müller, Martin Clochard, and David Basin. 2020. Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed Systems Verification. In *ACM Program. Lang. 4, OOPSLA, Article 152*. https://doi.org/10.1145/3428220
[55] Informal Systems. 2024. Apalache. https://apalache.informal.systems/.

[56] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. COBRA: Making Transactional Key-Value Stores Verifiably Serializable *(OSDI'20)*. USENIX Association, Article 4.
[57] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast In-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, 87–104.
[58] Shale Xiong, Andrea Cerone, Azalea Raad, and Philippa Gardner. 2020. Data Consistency in Transactional Storage Systems: A Centralised Semantics *(LIPIcs, Vol. 166)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 21:1–21:31.
[59] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *SOSP '15*. ACM, 263–278.
[60] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2018. Building Consistent Transactions with Inconsistent Replication. *ACM Trans. Comput. Syst.* 35, 4 (2018), 12:1–12:37.
[61] Jian Zhang, Ye Ji, Shuai Mu, and Cheng Tan. 2023. Viper: A Fast Snapshot Isolation Checker. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*. ACM, 654–671.

A THE PSEUDO CODE OF EIGER-PORT+

A.1 Logical Time

Logical time refers to a way of keeping track of events in a distributed system where there is no global clock available. It is based on the concept of logical clocks which are used to assign a unique timestamp to each event, ensuring that the order of events is preserved in a consistent and meaningful way. We employ a Lamport Clock (LC) [31], where each process maintains a local clock that is incremented whenever an event occurs. When a process sends a message, it includes its current timestamp, which is then used by the receiving process to update its clock. Lamport clocks are only partially ordered, meaning that they cannot always determine the exact ordering of events, but they can provide a causal ordering that is sufficient for many applications. Overall, logical time and Lamport clocks are essential tools for managing distributed systems and ensuring that they operate correctly and efficiently.

A.2 Client-side Procedures

Data-Structures.. Eiger-PORT+ keeps track of the global safe time via a timestamp called `gst`. Furthermore, it also stores the latest safe time for every partition in the map `lst_map`. The global safe time is the minimum of the local times as it represents the versions that are now safe to read from every partition.

Write Transactions. For every key-value pair, in parallel we call `prepare_write`, sending to the partition key, value, and client id (a unique identifier for every client). From this procedure, we receive back a prepared version, called `ver` and a timestamp `prepared_t`. Then, we compute the commit timestamp `commit_t` by selecting the largest prepared timestamp (line 21 in Figure 9). Finally, in parallel, we call the partition-side procedure `commit_write` for every key and value pair, providing the prepared version and commit timestamp. We receive as a response a new local safe time, which is used to update `lst_map` (line 23 in Figure 9).

Read Only Transactions. Firstly, we compute the current global safe time, by selecting the minimum of the local safe times (line 7 in Figure 9), this way we make sure it is safe to read at the global safe time from every server. Then, for every key, in parallel, we call the `read` procedure, sending to the partition in charge the key, the `gst`, and a client identifier. This procedure consequently returns the result and the new local safe time. Finally, we use the latter to update `lst_map`.

```
1  --- data structures:
2  lst_map[][]   --- maps key to its server's local safe time
3  gst           --- global safe time
4
5  --- procedures:
6  function read_only_txn (<keys>):
7    gst = min {lst_map.valueSet ()}
8    for k in keys --- in parallel
9      vals[k], lst = read (k, gst, cl_id)
10     lst_map[k] = lst
11   return vals
12
13 function write_txn (<keys, vals>):
14   --- Prepare
15   for k, v in <keys, vals> --- in parallel
16     ver, prepared_t = prepare_write (k, v, cl_id)
17     svr.msgs[k] = (ver, prepared_t)
18   --- all prepared_t received from involved servers
19
20   --- Commit
21   commit_t = max {svr_msgs.prepared_t}
22   for k, v in <keys, vals> --- in parallel
23     lst_map[k] = commit_write (svr_msgs[k].ver, commit_t)
24   return
```

Figure 9: Client-side procedures in Eiger-PORT+.

```
1  --- data structures:
2  lst           --- local safe time, updated upon writes
3
4  --- procedures:
5  function read (k, rts, cl_id):
6    ver = DS[k].at (rts)     --- vers are sorted by commit_t
7    for v in DS[k].greater_eq_commit_t_than (ver.commit_t)
8      if v.cl_id == cl_id    --- ensure read-your-writes
9        return v.val, lst
10   return ver.val, lst
```

Figure 10: Server-side read-only transactions in Eiger-PORT+.

A.3 Server-side Procedures

Data-Structures. In Eiger-PORT+ the partitions keep track of the local safe time `lst`, which, if there are any pending transactions in the partition, is the smallest of the pending timestamp, and otherwise is the current value of the Lamport clock. Moreover, we keep a list of timestamps, called `pending_wtxns`, containing all pending timestamps of uncommitted writes. Finally, the server contains a multi-versioned key-value store called `DS`, which is sorted by commit timestamps.

Read. Firstly, we fetch the version `ver` at the read timestamp `rts`, which is the global safe time according to the client that issued the read. This returns the newest version with a commit timestamp smaller than or equal to `rts`. Then, we find, if it exists, the newest committed version written by the same client, and return it on line 9 (Figure 10). Finally, if said version does not exist we return `ver` (line 10 in Figure 10). In either case, we also return the local safe time.

Prepare Write. The partition uses the current value of the Lamport clock as a timestamp `pending_t` for the pending transaction, it adds said value to `pending_wtxns` and advances the clock (lines 8-10 in Figure 11). Then, it adds a new version of the key to the database.

```
1  --- data structures:
2  lst            --- local safe time
3  pending_wtxns --- uncommitted write txns' pending timestamps
4  DS[][]         --- multi-versioned key-value store
5
6  --- procedures:
7  function prepare_write (k, v, cl_id):
8    pending_t = LamportClock.current ()
9    pending_wtxns.append (pending_t)
10   LamportClock.advance ()
11   ver = DS[k].create_new_ver (v, cl_id, pending_t)
12   ver.is_pending = true
13   return ver, LamportClock.current ()
14
15 function commit_write (ver, commit_t):
16   ver.commit_t = commit_t
17   ver.is_pending = false
18   pending_wtxns.remove (ver.pending_t)
19   if pending_wtxns is empty
20     lst = LamportClock.current ()
21   else
22     lst = pending_wtxns.head () --- min of pending_wtxns
23   return lst
```

Figure 11: Server-side write transactions in Eiger-PORT+.

The version contains the value, `pending_t`, the client id, and a flag `is_pending` to identify whether the version is pending, which is set to true (lines 11-12 in Figure 11). Finally, the procedure returns the version and the current value of the Lamport clock to the client.

Commit Write. Firstly, the commit timestamp of the version is set to `commit_t`, the `is_pending` flag is set to false and the version is removed from the list of pending transactions (lines 16-18 in Figure 11). Finally, `lst` is updated by selecting the smallest pending timestamp if there are any, or else by using the current Lamport clock value (lines 19-22 in Figure 11). The procedure then returns the new local safe time.

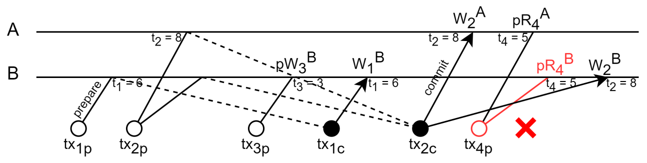

Figure 12: Counterexample showing atomic visibility violation in TAPIR based on the conference paper [59].

B TAPIR COUNTEREXAMPLES

In addition to the two counterexamples discussed in section 6, we provide a counterexample for atomic visibility in TAPIR, where we use the `tapir_occ_check` definition given in the conference paper [59]. The definition differs from the journal version [60] in line 4, replacing the abort condition $timestamp > MIN(prepared{-}writes[key])$ with $version < MIN(prepared{-}writes[key])$, where $timestamp$ is the transaction's proposed timestamp and $version$ is the read version's timestamp.

For the atomic visibility to hold, the read version's timestamp has to be larger than *all* prepared writes. Therefore aborting must not only happen when the read version's timestamp is lower than the minimum prepared writes' timestamps but when it's lower than any prepared write, i.e. maximum of prepared writes' timestamps. Otherwise, a transaction can read a version with a timestamp larger than the minimum but still smaller than some prepared writes' timestamps as shown in Figure 12.

Transaction tx_4 successfully prepares a read on key B because it reads W_1^B, so $version = 6$ while the minimum of prepared writes' timestamps are 3, which will not make the transaction abort. However, this execution demonstrates an atomic visibility violation because tx_2's writes are partially observed by tx_4.