VerIso: Verifiable Isolation Guarantees for Database Transactions

Shabnam Ghasemirad
ETH Zurich, Switzerland

Si Liu
ETH Zurich, Switzerland

Christoph Sprenger
ETH Zurich, Switzerland

Luca Multazzu
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 the first such framework, VerIso, developed within the theorem prover Isabelle/HOL. To showcase its use in verification, we model the strict two-phase locking concurrency control protocol and verify that it provides strict serializability isolation guarantee. Moreover, we show how VerIso helps identify isolation bugs during protocol design. We derive new counterexamples for the TAPIR protocol from failed attempts to prove its claimed strict serializability. In particular, we show that it violates a much weaker isolation level, namely, atomic visibility.

PVLDB Reference Format:
Shabnam Ghasemirad, Si Liu, Christoph Sprenger, Luca Multazzu, 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 progressed 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.

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) [22, 26, 43], many weaker isolation levels have emerged to cater for real-world applications. These include Read Atomicity (RA), supported by [5, 32, 34] and recently layered atop Facebook's TAO [12]; Transactional Causal Consistency (TCC), supported by [3, 9, 17, 34, 37, 38, 41, 51], and recently adopted by commercial databases [19, 42, 44].

Unfortunately, isolation bugs have repeatedly manifested themselves in carefully designed and heavily tested databases [8, 18, 23–25]. In particular, all these isolation bugs stemmed from design-level defects, rather than implementation errors. For example, YugabyteDB was found to violate SER as its conflict-detection mechanism failed to account for read conflicts [27], MariaDB's Galera cluster incorrectly claimed SI support [23, 30], and Lu et al. [39] recently reported violations of the claimed Strict SER (SSER) in TAPIR [58] and DrTM [55]. Eliminating such bugs, which requires costly redesigns, and achieving strong correctness guarantees is a crucial, challenging task for developers. Here, 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 [4, 16, 45, 53] has 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 [2, 7, 8, 11, 15, 36, 56], including the first characterization of SER via dependency graphs [7], an axiomatic framework based on abstract executions [11], and the recent operational semantics for isolation guarantees [56]. 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 a small number of executions, (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 [58].

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 unified coverage of a wide range of isolation guarantees. Rigor is guaranteed by mathematical proofs showing that *all* possible database behaviors with arbitrarily many processes and transactions satisfy the desired isolation guarantee.

To achieve this goal, we 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 [56], instantiatable by a range of prevalent isolation criteria, including SSER, SI, and TCC. We formalize this model, called the *abstract transaction model*, in Isabelle/HOL [47], an interactive theorem prover for higher-order

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

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 providing considerable proof automation and reducing the user's proof burden.

Within VeriSo, we formalize transaction protocol designs as transition systems, where different types of transitions correspond to different protocol steps. We establish their isolation guarantees by refinement of the appropriate instance of the abstract model, which ensures that the protocol's behaviors conform to the abstract model. 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 new transaction protocols at an early design stage. This mitigates the cost induced by redesign, implementation patching, and re-deployment of faulty designs.

To validate VeriSo, we analyze the isolation guarantees of two database designs. First, we model the classical Strict Two-Phase Locking (S2PL) protocol (combined with Two-Phase Commit) and we verify that it satisfies SSER. With this case study, we illustrate different aspects of VeriSo including protocol specification, refinement mapping, and refinement proof using invariants. Second, 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 [58] protocol, which claims to provide SSER. This protocol was previously found to exhibit a real-time ordering issue [39]. Using VeriSo, we discover a different counterexample, which violates *atomic visibility* [5], 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 the classical S2PL protocol combined with 2PC in VeriSo and prove that it satisfies SSER, thereby illustrating our protocol modeling and proof technique.
- We demonstrate how VeriSo can be used to find isolation bugs in database designs by examining failed proof obligations. In particular, for TAPIR, we construct novel counterexamples illustrating its violation of atomic visibility.

2 BACKGROUND

2.1 Isolation Levels

Distributed databases provide various isolation levels, depending on the system's desired 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 [5] and TCC [3, 37]. We will briefly explain some of these isolation guarantees.

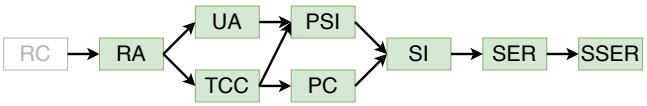

Figure 1: A hierarchy of prevalent isolation levels. RC: read committed [6]; RA: read atomicity [5]; UA: update atomicity [11, 35]; TCC: transactional causal consistency [3, 37]; PC: prefix consistency [10]; SI: snapshot isolation [6]; PSI: parallel SI [50]; SER: serializability [49]; SSER: strict SER [49]. $A \longrightarrow B$ means A is weaker than B. VeriSo covers the green levels.

Read Atomicity (RA). This is also known as *atomic visibility*, requiring that other transactions observe all or none of a transaction's updates. It prohibits *fractured reads*, such as in the scenario

$$T_1 : \mathtt{W}(A, \{B\}), \mathtt{W}(B, \{A\}) \qquad T_2 : \mathtt{R}(A, \{B\}), \mathtt{R}(B, \emptyset),$$

where Carol (in transaction T_2) only observes one direction of a new (bi-directional) friendship between Alice (A) and Bob (B) in a social network (recorded in T_1). RC allows such fractured reads as well as reading multiple versions of a key in one transaction.

Transactional Causal Consistency (TCC). This level requires that two causally related transactions appear to all clients in the same causal order. It prevents *causality violations*, such as

$$T_1 : \mathtt{W}(A, m) \qquad T_2 : \mathtt{R}(A, m), \mathtt{W}(B, r) \qquad T_3 : \mathtt{R}(A, \bot), \mathtt{R}(B, r),$$

where Carol (in T_3) observes Bob's response r to Alice's message m without seeing the message itself in a chatroom. Our notion of TCC also includes *convergence* [3, 37], which requires different clients to observe all transactions in the same total order. In practice, most causally-consistent databases provide convergence.

(Strict) Serializability ((S)SER). SER requires that the effects of every concurrent execution can also be achieved by a sequential execution. SSER additionally requires this sequential execution to preserve the real-time order of (non-overlapping) transactions.

2.2 Operational Semantics for Transactions

Xiong et al. [56] 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 3), called the *abstract transaction model*, which abstracts these KVSs into a single (centralized) multi-versioned KVS $\mathcal{K} : \mathtt{key} \longrightarrow \mathtt{list(version)}$ 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 (the transactions that have read this version).

The reader set tracks write-read (WR) dependencies. In a real, distributed system, each client cl has a different partial *client view* of $\mathcal{K}$, and this is modeled by explicitly representing these views in the model as mappings $\mathcal{U}(cl) : \mathtt{key} \longrightarrow \mathcal{P}(\mathbb{N})$, describing, for each key, the set of versions (denoted by list indices) visible to the client.

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 RA as the model's baseline isolation guarantee. Transactions are described by a *fingerprint* $\mathcal{F} : \texttt{key} \times \{\texttt{R},\texttt{W}\} \rightharpoonup \texttt{value}$, 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 KVS. 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 (Figure 1).

2.3 Isabelle/HOL and Notation

We use the Isabelle/HOL proof assistant for our modeling and proofs [46, 48]. Isabelle is a generic framework for implementing logics, and Isabelle/HOL is its instance to higher-order logic (HOL). HOL can be paraphrased as "functional programming plus quantifiers". Isabelle offers powerful automated reasoning tools, including an integration of various external (first-order logic) automated theorem provers. We chose Isabelle/HOL for its high expressiveness and good proof automation, which allows us to naturally formalize the framework's structures and obtain strong protocol correctness guarantees for arbitrary numbers of processes and transactions.

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 $x \notin dom(g)$. We also define $A_\bot = A \uplus \{\bot\}$. For a relation $R \subseteq A \times A$, R^{-1} denotes its converse and R^+ its transitive closure.

3 SYSTEM MODELING AND REFINEMENT

We introduce the modeling and verification formalism [1, 40] we use in our work and illustrate it with a simple database example.

3.1 System Models

We use *labeled transition systems* (LTS) to model both 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 (Atomic Transactions). We model a simple centralized database system with a single-version KVS and atomic (one-shot) transactions as an LTS $\mathcal{A}$. A state of our LTS $\mathcal{A}$ is a KVS, which we represent as a function $\mathcal{S} : \texttt{key} \to \texttt{value}$, from keys to values. All states are initial states. Besides skip, the LTS $\mathcal{A}$ has a single (parametrized) event $\texttt{txn}(f_R, f_W)$, where the parameters $f_R, f_W : \texttt{key} \rightharpoonup \texttt{value}$ respectively denote a read and a write fingerprint. We specify the associated transition relation as follows:

$$\texttt{txn}(f_R, f_W) : \; f_R \subseteq \mathcal{S} \;\wedge\; \mathcal{S}' = \mathcal{S} \triangleright f_W,$$

meaning that $\mathcal{S} \xrightarrow{\texttt{txn}(f_R,f_W)} \mathcal{S}'$ is defined by the predicate after the colon. Here, $f_R \subseteq \mathcal{S}$ is the guard, requiring that the values read are those in the KVS $\mathcal{S}$, and $\mathcal{S}' = \mathcal{S} \triangleright f_W$ is the update, stating that the KVS $\mathcal{S}'$ after the transition equals the KVS $\mathcal{S}$ updated with the values in f_W. The updated KVS $(\mathcal{S} \triangleright f_W)(k)$ is defined by $f_W(k)$ if $k \in dom(f_W)$ and $\mathcal{S}(k)$ otherwise. Note that the parameter f_R can be considered as an output (the values read) and f_W as an input (the values to be written) to the event. Hence, in this very abstract model $\mathcal{A}$, all of a transaction's reads and writes are executed in a single atomic state transition.

3.2 Invariants and Refinement

We define invariants as supersets of the set of reachable states (or, equivalently, predicates holding on all reachable states), and refinement as a process for successively adding features to models.

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 $\text{reach}(\mathcal{E})$. A set of states J is an *invariant* if $\text{reach}(\mathcal{E}) \subseteq J$. 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* $r : S_2 \to S_1$ and $\pi : E_2 \to E_1$ 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 for all states $s, s' \in S_2$,

(i) $r(s) \in I_1$, and
(ii) $r(s) \xrightarrow{\pi(e)}_1 r(s')$ whenever $s \xrightarrow{e}_2 s'$.

We can assume without loss of generality that the state s in these proof obligations is reachable. This allows us to use invariants of the concrete model in refinement proofs, which is often necessary. When using guards and updates, and an invariant J of $\mathcal{E}_2$, (ii) reduces to two proof obligations: assuming $G^2_e(s)$ and $s \in J$, 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*).

Guard strengthening ensures that whenever the concrete event is executable then so is the corresponding abstract event. Update correspondence expresses that the refinement mapping r commutes with the concrete and abstract state updates.

Refinement guarantees the inclusion of sets of reachable states (modulo r), i.e., $r(\text{reach}(\mathcal{E}_2)) \subseteq \text{reach}(\mathcal{E}_1)$. Hence, $\mathcal{E}_1$ specifies which states $\mathcal{E}_2$ is allowed to reach (modulo r).

Example 2 (Refined transactions). We define a more concrete model C. Instead of executing transactions in a single atomic transition, we have three operations, read, write, and commit, which we model as separate events. This model's states consist of triples $(\mathcal{S}, \mathcal{F}_R, \mathcal{F}_W)$, where $\mathcal{S}$ is as before, and $\mathcal{F}_R$ and $\mathcal{F}_W$ are read and write fingerprints, which are now part of the state. Initial states are those with empty fingerprints. The model C's three events and

their transition relations are as follows.

$$\begin{aligned}
\texttt{read}(k,v):&\ \mathcal{S}(k) = v \wedge \mathcal{F}'_R = \mathcal{F}_R[k \mapsto v]\\
\texttt{write}(k,v):&\ \mathcal{F}'_W = \mathcal{F}_W[k \mapsto v]\\
\texttt{commit}(f_R,f_W):&\ f_R = \mathcal{F}_R \wedge f_W = \mathcal{F}_W \wedge\\
&\ \mathcal{S}' = \mathcal{S} \triangleright \mathcal{F}_W \wedge \mathcal{F}'_R = \emptyset \wedge \mathcal{F}'_W = \emptyset
\end{aligned}$$

The read event's guard checks that the KVS $\mathcal{S}$ maps the key k to the value v and updates the read fingerprint $\mathcal{F}_R$ with this mapping. The write event updates the write fingerprint $\mathcal{F}_W$ with the given key-value mapping. Finally, the commit event's guard just binds the current fingerprints to the respective event parameters. This event updates the KVS $\mathcal{S}$ with the write fingerprint $\mathcal{F}_W$ as in Example 1 and resets the fingerprints to the empty maps. Hence, this model only allows the *serialized* execution of transactions, one at a time.

Example 3 (Refinement proof). We now prove $C \preccurlyeq_{r,\pi} \mathcal{A}$ for suitable refinement mappings r and π. We define the state mapping r by $r(\mathcal{S}, \mathcal{F}_R, \mathcal{F}_W) = \mathcal{S}$ and the event mapping π as $\pi(\texttt{commit}(f_R, f_W)) = \texttt{txn}(f_R, f_W)$, with all other events mapping to skip. This means that only the commit event has an observable effect on the abstract state and this effect corresponds to an abstract transaction.

For the refinement proof, note that the proof obligations for the read and write events trivially hold as these events do not change $\mathcal{S}$. The interesting case is the commit event. It is easy to see that update correspondence holds, since the concrete commit and the abstract `txn` events' KVS updates are identical. For the guard strengthening proof obligation, we must show that $f_R = \mathcal{F}_R \wedge f_W = \mathcal{F}_W$ implies $f_R \subseteq \mathcal{S}$. This is not possible, as the concrete guard provides no information about the KVS $\mathcal{S}$. However, inspecting the concrete system C, we see that the read event only records key-value mappings in $\mathcal{F}_R$ that are also present in $\mathcal{S}$. Hence, we can establish $\mathcal{F}_R \subseteq \mathcal{S}$ as invariant of C. We can then use this invariant together with the guard $f_R = \mathcal{F}_R$ to complete the refinement proof.

4 THE VERISO FORMAL FRAMEWORK

Our framework is centered around our formalization of the abstract transaction model introduced in Section 2.2. We formalize the abstract model in Isabelle/HOL as an LTS $\mathcal{I}(\mathit{IL})$, parameterized by an isolation level $\mathit{IL} \in \{\texttt{RA}, \texttt{UA}, \texttt{TCC}, \texttt{PSI}, \texttt{CP}, \texttt{SI}, \texttt{SER}, \texttt{SSER}\}$ (cf. Figure 1). We describe this formalization in Section 4.1.

Given a pseudocode specification of a concurrency control protocol design, its modeling and verification using our framework consists of the following three steps:

(1) Formalize the pseudocode specification of the protocol as an LTS model $\mathcal{M}$ in Isabelle/HOL, as discussed in Section 4.2.
(2) Specify the protocol's desired isolation level IL by the appropriate instance, $\mathcal{I}(\mathit{IL})$, of the abstract transaction model.
(3) Define refinement mappings on states r and events π and prove that $\mathcal{M}$ refines the abstract transaction model instance $\mathcal{I}(\mathit{IL})$. This is explained in Section 4.3.

4.1 Formalizing the Abstract Transaction Model

4.1.1 Parametrized abstract transaction model. The model's states $(\mathcal{K}, \mathcal{U})$, called configurations, consist of a central KVS $\mathcal{K}$ and the client views $\mathcal{U}$, mapping client IDs to their views, as explained in Section 2.2. This structure is formalized in Figure 2. In particular, we model transaction identifiers as a datatype `txid`, whose elements $\texttt{Tn}(sn, cl)$ are indexed by the issuing client cl and a sequence number sn, and we model versions as a record type `version` in Isabelle/HOL.

```
datatype txid = Tn(nat, cl_id)

record version =
  v_value : value
  v_writer : txid
  v_readerset : set(txid)

type_synonym kv_store = key ⇀ list(version)
type_synonym view = key ⇀ set(nat)
type_synonym config = kv_store × (cl_id ⇀ view)
```

Figure 2: Configurations of abstract transaction model.

The model has three events: an (atomic) commit event, a view extension event, and the idling event. The commit event's guard depends on IL, which is specified by two elements:

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

We will further describe R_{IL} and $\texttt{vShift}_{\mathit{IL}}$ below.

The abstract commit event's transition relation

$$(\mathcal{K}, \mathcal{U}) \xrightarrow{\texttt{commit}(cl,sn,u,\mathcal{F})}_{\mathit{IL}} (\mathcal{K}', \mathcal{U}')$$

is defined in Isabelle/HOL by:

```
definition commit(cl, sn, u, F, (K, U), (K', U')) ⟷
  U(cl) ⊑ u ∧ wf(K, u) ∧ wf(K', u') ∧    -- basic view guards
  LWW(K, u, F) ∧                          -- last-write-wins
  Tn(sn, cl) ∈ nextTxids(K, cl) ∧         -- txid freshness
  canCommit(K, u, F, R_IL) ∧ vShift_IL(K, u, K', u') ∧ --IL-specific
  K' = UpdateKV(K, Tn(sn, cl), u, F) ∧ U' = U[cl ↦ u']  -- updates
```

The transition updates the configuration $(\mathcal{K}, \mathcal{U})$ to the new configuration $(\mathcal{K}', \mathcal{U}')$, where $\mathcal{K}'$ is the updated KVS and $\mathcal{U}'$ updates the client cl's view to u'. In particular, $\mathcal{K}'$ is obtained from $\mathcal{K}$ by recording the operations described by the fingerprint $\mathcal{F}$, i.e., the writes append a new version with writer ID $\texttt{Tn}(sn, cl)$ to the respective key's version list, and the reads add $\texttt{Tn}(sn, cl)$ to the respective versions' reader sets. The guards have the following meanings:

- $\mathcal{U}(cl) \sqsubseteq u$ allows one to extend the client cl's current view to a (point-wise) larger one before committing.
- $\texttt{wf}(\mathcal{K}, u)$ and $\texttt{wf}(\mathcal{K}', u')$ require u and u' to be *wellformed views*, i.e., atomic and holding indices that point to existing versions.
- $\texttt{LWW}(\mathcal{K}, u, \mathcal{F})$ captures the *last-write-wins* conflict resolution policy, where a client reads each key's latest version in its view u. If $\mathcal{F}(k, \texttt{R}) = v$ then v must be the version $\mathcal{K}(k, i)$'s value, where i is the highest index in $u(k)$.
- $\texttt{Tn}(sn, cl) \in \texttt{nextTxids}(\mathcal{K}, cl)$ represents the transaction ID freshness requirement, i.e., the sequence number sn is larger than any of the client cl's sequence numbers used in $\mathcal{K}$.
- $\texttt{canCommit}(\mathcal{K}, u, \mathcal{F}, R_{\mathit{IL}})$ is the central commit condition, which ensures that it is safe to commit a transaction at a given isolation level IL. It requires that the set of visible transactions $\texttt{visTx}(\mathcal{K}, u)$ (i.e., the writers of the versions that the view u points to) is *closed* under the relation R_{IL} in the sense that

$$(R_{\mathit{IL}}^{-1})^+(\texttt{visTx}(\mathcal{K}, u)) \subseteq \texttt{visTx}(\mathcal{K}, u) \cup \texttt{rdonly}(\mathcal{K}). \quad (1)$$

In other words, following the relation R_{IL} backwards from visible transactions, we only see visible or read-only transactions.
- $\texttt{vShift}_{IL}(\mathcal{K}, u, \mathcal{K}', u')$ constrains the allowed modifications of the client view during the commit. This predicate captures session guarantees like monotonic reads and read-your-writes.

We define the view extension event's transition relation

$$(\mathcal{K}, \mathcal{U}) \xrightarrow{\texttt{xview}(cl,u)}_{IL} (\mathcal{K}', \mathcal{U}')$$

in Isabelle/HOL by:

```
definition xview(cl, u, (𝒦, 𝒰), (𝒦', 𝒰')) ⟷
  𝒰(cl) ⊑ u ∧ wf(𝒦, u) ∧
  𝒦' = 𝒦 ∧ 𝒰' = 𝒰[cl ↦ u]
```

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

4.1.2 Instantiation to concrete isolation levels. To instantiate this abstract model for a particular isolation level IL, one specifies the relation R_{IL} and the predicate $\texttt{vShift}_{IL}$. We describe here the definitions for $IL \in \{\texttt{RA}, \texttt{TCC}, \texttt{SSER}\}$ and we refer the reader to [56] for other instantiations. For the model's baseline isolation level RA, the conditions canCommit (with $R_{\texttt{RA}} = \emptyset$) and $\texttt{vShift}_{\texttt{RA}}$ always hold. For TCC, we define $R_{\texttt{TCC}} = \texttt{SO} \cup \texttt{WR}_{\mathcal{K}}$, where SO captures the clients' session orders and $\texttt{WR}_{\mathcal{K}}$ is the write-read dependency between transactions in $\mathcal{K}$, determined by the reader set associated to each version. The resulting commit condition $\texttt{canCommit}_{\texttt{TCC}}$ requires that the views are closed under *causal dependencies*. The predicate $\texttt{vShift}_{\texttt{TCC}}$ captures the monotonic reads and read-your-writes session guarantees. For SSER, we have $R_{\texttt{SSER}} = \texttt{WW}_{\mathcal{K}}^{-1}$, where $\texttt{WW}_{\mathcal{K}}$ is the write-write dependency (i.e., per-key version order) on transactions in $\mathcal{K}$. Hence, the commit condition $\texttt{canCommit}_{\texttt{SSER}}$ expresses that the views must include *all* versions in $\mathcal{K}$ before the commit. The condition $\texttt{vShift}_{\texttt{SSER}}$ is true, i.e., it always holds.

Note that the semantics' built-in atomic visibility precludes the representation of the RC isolation level (cf. Figure 1). In future work, we will extend the model to also cover this level.

4.1.3 Discussion of formalization. The formalization involved both design choices and adaptations compared to [56]. 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 (1) 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 [56] 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.

4.2 Protocol Modeling

We assume that protocol designs are given as pseudocode specifications for clients and servers. We explain how to transform such specifications into a formal LTS protocol model in Isabelle/HOL.

```
function cl_compute(tasks):              -- client
  for each server svr:
    results(svr) = svr_compute(tasks(svr), svr)
  return aggregate(results)

function svr_compute(task, svr):         -- servers
  result = compute(task)
  return result
```

Figure 3: Pseudocode for remote computation.

```
datatype cl_state =      -- client state
  cl_idle | cl_invoked(tasks) | cl_done(final)

record cl_conf =         -- client configuration
  cl_state : cl_state

datatype comp_state =    -- computation state
  svr_idle | svr_done(result)

record svr_conf =        -- server configuration
  svr_state : cl_id → comp_state

record global_conf =     -- global configuration
  cls : cl_id → cl_conf
  svrs : svr_id → svr_conf
```

Figure 4: Client, server, and global configurations.

For illustration, we use a simple remote computation example, where clients outsource computation tasks to servers using (asynchronous) remote procedure calls (RPCs) and then collect and aggregate the servers' results into a final result (Figure 3). The same modeling technique also applies to concurrency control protocols. We discuss some points specific to those protocols at the end of this subsection. Formalizing a protocol design as an LTS in Isabelle/HOL requires the definition of the LTS' global configurations and events.

4.2.1 Protocol configurations. Protocol configurations are composed of the clients' and the servers' local configurations (Figure 4). A client's configuration consists of the client's control state (of type cl_conf), which here may be idle, invoked with given tasks, or done with the final computation result. The server has a computation state (of type comp_state) per request. We assume for simplicity that each client only invokes a server once, so we can identify requests with clients. A computation's state is either idle or done with a result.

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 and cl_state are record field projections. For readability, we will write this as cl_state(cl) when s is clear from the context. We will use cl_state'(cl) for the field's value in the successor state s' of an event. Similar shorthands apply to all record fields.

4.2.2 Protocol events. We model protocol events according to the following principles. Events only change either a single client or a single server's configuration. This ensures that clients and servers can be seen as independent components with interleaved events. To communicate, these components directly access each other's configurations. More precisely, the destination (or receiver) component reads the desired information from the data owner (or sender). The

```
1  -- client-side events
2  definition cl_compute_invoke(cl, tasks) ⟷
3    cl_state(cl) = cl_idle ∧
4    cl_state'(cl) = cl_invoked(tasks)
5
6  definition cl_compute_response(cl, results, final) ⟷
7    cl_state(cl) = cl_invoked(_) ∧
8    (∀svr. svr_state(svr, cl) = svr_done(results(svr)) ∧
9    final = aggregate(results) ∧
10   cl_state'(cl) = cl_done(final)
11
12 -- server-side event
13 definition svr_compute(svr, cl, task) ⟷
14   svr_state(svr, cl) = svr_idle ∧
15   (∃tasks. cl_state(cl) = cl_invoked(tasks) ∧
16            task = tasks(svr)) ∧
17   svr_state'(svr, cl) = svr_done(compute(task))
```

Figure 5: Events for remote computation example.

sender just provides the information, but does not actively communicate itself (e.g., send a message). This is a standard abstraction in protocol modeling, which can later be refined into explicit message passing through a channel. Events may have parameters, which correspond to the free variables in the event specification.

We translate the protocol's pseudocode from Figure 3 into two client events, one for the RPC request and the other for the response, and one server event, which handles the entire RPC on the server side (Figure 5). The client event `cl_compute_invoke` transits from idle to the invoked state, storing the given `tasks`. Note that no client-server communication is modeled in this event. The server event `svr_compute` reads its task `tasks(svr)` from client `cl`'s invoked state, which models receiving a message containing `tasks(svr)` from `cl`. The existential quantifier on Line 15 means that the `tasks` map itself is hidden from the server, i.e., not transmitted. Finally, the client event `cl_compute_response` starts in an invoked state, where the tasks stored do not matter, as indicated by the underscore (Line 7), collects the results from all servers into the `results` map (Line 8), computes the `final` result (Line 9), and stores it in its new done state (Line 10). Note that record fields not mentioned in an event's update remain unchanged.

4.2.3 Modeling concurrency control protocols. We consider distributed systems with per-client transaction coordinators 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 concurrency control and do not model (orthogonal) replication aspects.

Most concurrency control protocols follow a two-phase commit structure, which is reflected in the client's and the server's states. We assume the clients perform transactions sequentially, whereas the servers handle an arbitrary number of concurrent transactions. Hence, the servers will have per-transaction states. Both client and server configurations may contain additional fields to store protocol-specific information such as clocks, timestamps, or version lists.

4.3 Protocol Verification

After modeling a concurrency control protocol, we prove by refinement that it implements the desired instance of the abstract transaction model. We show how to define refinement mappings relating the protocol model to the abstract one and conduct the refinement proof. We also briefly discuss an advanced proof technique for the case when refinement alone is insufficient to prove correctness.

4.3.1 Defining the refinement mappings. To relate the protocol and abstract models, we must define the refinement mappings r on states and π on events. To define π, we identify one or more protocol events that correspond to the protocol's transaction commit points and we map these events to the abstract commit event. These are usually client-side (i.e., coordinator) commit events. Events that change a client's view of the KVS (e.g., by making additional versions visible) refine the abstract view extension event. All remaining events must refine the abstract idling event (skip), meaning they have no corresponding effect in the abstract model.

To define r, we reconstruct an abstract configuration $r(s)$ from a concrete protocol configuration s, i.e.,

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

The function $\mathcal{K}\_\texttt{of}(s)$ 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 $\mathcal{U}\_\texttt{of}(s)$ reconstructs the abstract configuration's client views from the protocol configurations. For example, in some protocols for achieving TCC, clients maintain an explicit threshold (e.g., a timestamp) for reading "safe" versions, which are guaranteed to be committed. In this case, the view would include all versions below that threshold. In other protocols (e.g., for achieving SSER), clients must read the latest version of each key. Here, one can use full views of each key, which include all versions. This part of r is mostly constrained by the need to satisfy the view-related guards of the abstract commit event (wellformedness, `LWW`, `canCommit`, `vShift`) when refining the protocol's committing events.

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. However, they are not required for the protocol execution itself. In particular, event guards must not depend on them.

4.3.2 Refinement proofs. To establish the desired refinement

$$\mathcal{M} \preccurlyeq_{r,\pi} \mathcal{I}(\mathcal{IL}), \tag{2}$$

we must prove for each concrete event (i) guard strengthening, i.e., that the protocol event's guards imply the abstract event's guards, and (ii) update correspondence, i.e., that the abstract and concrete updates commute with the refinement mapping. VeriIso provides proof rules that support such refinements. Recall that the refinement (2) implies the following *correctness condition*:

$$r(\mathsf{reach}(\mathcal{M})) \subseteq \mathsf{reach}(\mathcal{I}(\mathcal{IL})), \tag{3}$$

where r is the state component of the refinement mapping.

We typically approach such proofs in the following way. We first prove the easy cases of concrete events that map to the abstract skip event. For these, guard strengthening is trivial and update correspondence shows that the concrete event has no effect on the abstract state. More interesting are the events refining the abstract commit event (and also view extension). Here, update correspondence ensures that the protocol's concrete KVS updates are consistent with the abstract centralized KVS. Most of the work goes into proving guard strengthening, i.e., establishing the abstract commit's guards from the concrete event's guards. By first proving the basic guards, i.e., view extension ($\mathcal{U}(cl) \sqsubseteq u$) and view wellformedness, followed by LWW, we know that the protocol satisfies the baseline RA isolation level. We then establish the guards `canCommit` and `vShift`, which are specific to the desired isolation level.

In case we are unable to establish a proof obligation, most often an abstract commit guard, this may happen for different reasons:

1. We may discover a *mistake* in the protocol modeling or in the definition of the refinement mapping, which we must correct.
2. We may *miss some information* to complete the proof. In this case, we try to prove a protocol *invariant* providing that information. For example, Isabelle's proof state may indicate a prepared client and a corresponding committed server, which is a contradiction derivable from a suitable invariant relating client and server states. Below, we list some typical invariants.
3. If neither of the previous two cases applies, further analysis may indicate a *counterexample* violating the desired isolation level. We will explain this case in Section 6. Note that a counterexample may also arise from a failed attempt to prove a required protocol invariant.

This is an iterative process that continues until all proof obligations are established or a counterexample is found.

4.3.3 Protocol invariants. Refinement proofs invariably require 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 related to particular protocol mechanisms such as locking (cf. Section 5) or timestamps.

4.3.4 Advanced verification technique. Sometimes, refinement is insufficient to establish the correctness condition in Equation (3). In particular, protocols relying on *timestamps* to isolate transactions, such as those commonly seen for *optimistic* concurrency control or for achieving TCC isolation, usually use the timestamps to define an order on versions and to identify "safe-to-read" versions (defining 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. However, for such protocols, the *execution order* of commits and the *order of the associated commit timestamps* may differ and

```
-- client-side (transaction manager)
function initiateTransaction(transaction):
  for each RM in transaction.participants:   -- 2PC: prepare
    resp[RM] = prepare(transaction, RM)
  if all participants respond with "okay":
    for each RM in transaction.participants: -- 2PC: commit
        commit(transaction, RM)
  else: -- abort if any RM responds "not okay"
    for each RM in transaction.participants:
        abort(transaction, RM)

-- server-side (resource manager)
function prepare(transaction, RM):
  try:
    acquireLocks(transaction.resources[RM])  -- S2PL: grow
    performOperations(transaction.operations[RM])
    return "okay"
  except:
    return "not okay"

function commit(transaction, RM):
  commitTransaction(transaction)       -- finalize changes
  releaseLocks(transaction.resources[RM])  -- S2PL: shrink

function abort(transaction, RM):
  abortTransaction(transaction)        -- rollback changes
  releaseLocks(transaction.resources[RM])  -- S2PL: shrink
```

Figure 6: S2PL pseudocode for client and server sides.

executions with such *inverted commits* may thus require *inserting* rather than *appending* a key's new version to its version list. Since the abstract model always appends new versions to the version lists, a refinement proof alone would fail for such executions.

Proving the correctness condition in Equation (3) for such protocols therefore requires the use of an advanced proof technique based on Lipton's reduction method [31] to reorder and eliminate inverted commits prior to the refinement proof. VerIso also provides proof rules for such reduction proofs. In [20], we explain this technique in more detail and we apply it to a protocol achieving TCC.

5 STRICT TWO-PHASE LOCKING

To illustrate VerIso's application, we analyze a well-known distributed concurrency control protocol, namely Strict Two-Phase Locking (S2PL), which is commonly employed in settings with strong isolation and reliability requirements. 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 locks for the involved keys are available. We assume that the failure to acquire any of these locks results in aborting the transaction (known as the "no-wait" method). This protocol is usually combined with the Two-Phase Commit (2PC) protocol to achieve atomicity by ensuring that all servers involved in the transaction are prepared before committing. Figure 6 shows the S2PL pseudocode from which we next construct a formal LTS protocol specification. We then verify in VerIso that it satisfies SSER.

5.1 Protocol Specification

We first analyze the pseudocode's communication structure. The client broadcasts two (asynchronous) RPC requests to all servers involved in a transaction: (i) a prepare request, and (ii) a commit

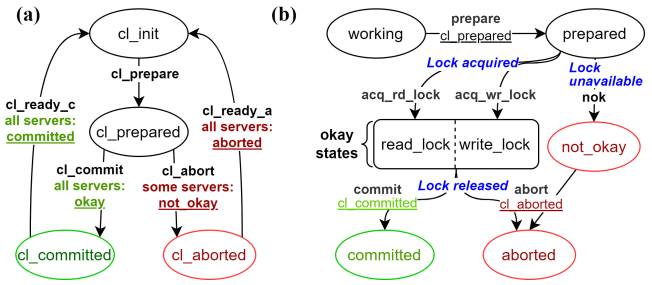

Figure 7: S2PL: state diagrams of (a) a client's `cl_state` and (b) a server's `svr_state` for a given transaction.

Table 1: Mapping RPC requests and responses to LTS events.

RPC	client event	server event
prepare request	`cl_prepare`	`prepare`
prepare response	`cl_commit` or `cl_abort`	`acq_rd_lock` or `acq_wr_lock` or `nok`
commit request	`cl_commit`	`commit`
commit response	`cl_ready_c`	`commit`
abort request	`cl_abort`	`abort`
abort response	`cl_ready_a`	`abort`

or abort request depending on the prepare responses. The servers send two RPC responses to the client: (i) an "okay" or "not okay" response to prepare, and (ii) a "committed" or "aborted" response.

We then translate this structure into control state diagrams of (a) a given client `cl`'s state and (b) a server's state for a given key `k` and transaction `t` (Figure 7). The state transitions are labeled with event names (in bold) and the transition guards (underlined). The RPC requests and responses are each represented in the diagrams by an event on the sender and on the receiver side (Table 1).

Recall from Section 4.2 that we model communication by the "receiver" reading the desired information from the configuration of the (passive) "sender". Note that we can often combine receiving a message and sending the next message into one event. For example, the client commit event checks the servers' prepare responses and transitions to the `cl_committed` state (i.e., "sends" a commit message). Similarly, the server side commit and abort events handle both the RPC request and response. In contrast, the server prepare event only receives a prepare request and the response is provided by a lock-acquiring event or the `nok` event. Next, based on the diagrams of Figure 7, we define the configurations and the events of our formal LTS protocol model of S2PL, called `TPL`, in Isabelle/HOL.

5.1.1 Configurations. We model the client, server, and global configurations as Isabelle/HOL records (Figure 8). The global configuration consists of mappings from client IDs 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, with one of four values shown in Figure 7a. The sequence number keeps track of a client's current transaction. Each server responsible for a given key `k`, called server `k` for short, stores 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

```
datatype txn_state =      -- transaction state
  cl_init | cl_prepared | cl_committed | cl_aborted

record cl_conf =          -- client configuration
  cl_state : txn_state
  cl_sn : nat

datatype ver_state =      -- version state
  working | prepared | read_lock | write_lock |
  not_okay | committed | aborted

record svr_conf =         -- server configuration
  svr_state : txid ⇀ ver_state
  svr_vl : list(version)
  svr_fp : txid × {R,W} ⇀ value

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

Figure 8: S2PL client, server, and global configurations.

in one of the seven states depicted in Figure 7b and its fingerprint `svr_fp(k,t)` on `k`, maps read and write operations to a value, if any.

5.1.2 Events. The client and server events change their respective transaction states as described in Figure 7. 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. Once a server `k` is in the prepared state for some transaction `t`, it tries to acquire a read or write lock for `k`, depending on the intended operations. Here is the definition of the server event `acq_write_lock`:

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

This event acquires a write lock for transaction `t` to update the key `k` with the value v_w and possibly also for reading the key's latest value v_r, i.e., v_r is either $\bot$ (no read) or `last_ver_v (svr_vl k)` (read) (Line 5). To execute the event, its guards require that (i) the server `k` is in the `prepared` state (Line 3), and (ii) no transaction has already acquired a lock on `k` (Line 4). The event updates its state to `write_lock` (Line 7) and records its reads and writes in its key fingerprint variable `svr_fp` (Line 8).

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

```
1 definition cl_commit(cl, sn, u, F) ⟷
2   -- guards:
3   cl_state(cl) = cl_prepared ∧
4   ∀k. dom(F(k)) ≠ {} ⟶ is_locked(svr_state(k, get_txn(cl))) ∧
5   sn = cl_sn(cl) ∧
6   u = λk. full_view(𝒦_of_TPL(s, k)) ∧
7   F = λk. svr_fp(k, get_txn(cl)) ∧
8   -- updates:
9   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 involved servers are in a 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 abstract KVS $\mathcal{K}$_of_TPL(s), which includes all versions of all keys (see Section 5.2), 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. This includes the values of the keys read in the transaction. 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.

5.2 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 $\mathcal{I}$(SSER).

Theorem 4. $\mathsf{TPL} \preccurlyeq_{r_{\mathsf{TPL}},\pi_{\mathsf{TPL}}} \mathcal{I}(\mathsf{SSER})$.

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

5.2.1 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 all key servers' version lists, extended with the effects of ongoing transactions' *client-committed* operations (cf. Section 4.3).

$\mathcal{U}$_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.

5.2.2 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 $\mathcal{I}$(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 4.1) into guard strengthening, i.e., that the concrete event's guards imply the abstract guards, and update correspondence (cf. Section 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$(\mathcal{K}, \mathsf{u}, \mathsf{F}, R_{\mathsf{SSER}})$ 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. VeriIso provides a general, reusable lemma showing that this is the case for full views. (ii) The guard LWW$(\mathcal{K}, \mathsf{u}, \mathsf{F})$ 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 VeriIso's library lemmas about abstract state updates.

5.2.3 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 4.3.3, 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 locking scenario (read or write) to the expected fingerprints on key servers.

6 FINDING ISOLATION BUGS USING VERISO

In this section, we demonstrate how VeriIso can also help uncover isolation bugs in protocols. We begin with an overview of our approach and then present a case study on the TAPIR protocol [58], where we uncover a previously unknown atomic visibility violation.

6.1 Approach

We discover isolation bugs from failed protocol verification attempts. Hence, we start by setting up a refinement proof for the claimed isolation guarantee in VeriIso (cf. Section 4.3). If we are unable to prove a given proof obligation or required invariant, even after potential improvements to the model and the refinement mappings, this may indicate a problem with the protocol. We then analyze the situation further and try to construct a counterexample protocol execution that violates the desired isolation guarantee.

In the following, we consider different cases in such an analysis, focusing on the most interesting case, namely, the failure to establish guard strengthening for events that were intended to refine the abstract commit event. We start from the basic abstract guards, shared by all isolation levels, and progress to the more complex ones, specific to a given isolation level. For each category of guards, we assume the preceding categories' guards to be already proven. Our step-by-step approach of breaking down such proofs into small and manageable proof obligations, corresponding to different protocol aspects, facilitates spotting isolation bugs in protocol designs.

6.1.1 Basic: View wellformedness and view extension. If a wellformedness guard fails because of view atomicity, either the protocol inherently does not support atomic views, which immediately shows a violation of RA, or the refinement mapping must be updated to include all versions written by a given transaction in the view. Otherwise, if the guard fails due to indices pointing to non-existent versions, again the refinement mapping's view construction must be checked. The view extension guard's failure indicates that some

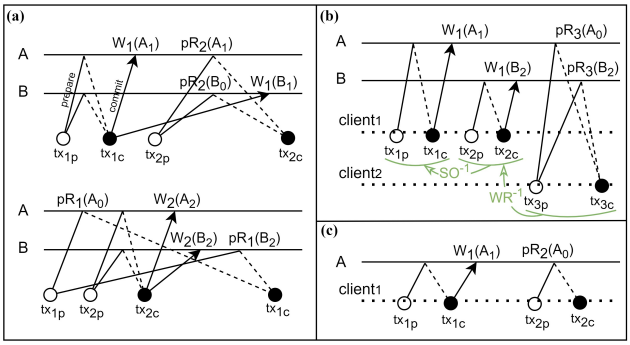

Figure 9: Interleaving of transactions' events, violating (a) LWW, (b) canCommit$_{TCC}$, and (c) vShift$_{RYW}$. Clients' *prepare* and *commit* events are shown by the tx_p and tx_c nodes. Servers' prepare-read and commit-write events are marked by pR$_{txid}$(read_version) and W$_{txid}$(written_version) on keys.

versions, previously visible to the client, are removed from its view, which can be directly checked in the protocol definition or again indicates a need for updating the refinement mapping.

6.1.2 For RA: Last-write-wins (LWW). When LWW fails, a client might not read the latest version in its view. An important special case of this occurs when the client reads a transaction's update on one server but reads an older version on another server that was also updated by that transaction, despite having both updates in its view (as view atomicity is assumed proven). This behavior corresponds to the fractured read anomaly, which indicates the violation of RA.

To construct an execution that captures this failure, at least two transactions are required: A write transaction, writing to at least two keys, and a read transaction that does not read the latest version on at least one of those keys. Figure 9a shows two possible interleavings of such transactions, both violating the LWW condition and exhibiting fractured reads.

6.1.3 For higher isolation levels: canCommit *and* vShift$_{IL}$. If all previous proof obligations are proven, a failure in these guards can indicate an isolation bug for the given IL. In these cases, similar to the LWW case, we must construct executions that capture these failures. For canCommit, by starting from an interleaving of transactions, and considering a client's view u, which defines the set of transactions visible to it (visTx($\mathcal{K}, u$)), we must repeatedly apply R_{IL}^{-1} to this set until the result contains an *invisible* transaction, i.e., a transaction that is neither read-only nor in u. An example of such an execution for IL = TCC is shown in Figure 9b. For vShift, the session guarantees (monotonic reads or read-your-writes) fail. Therefore, the execution should demonstrate the failure of these guarantees. Figure 9c shows an example execution for violating read-your-writes.

After building a candidate execution that violates a proof obligation, we might need to assign some values, e.g., timestamps, or add transactions, such that the protocol can run the given execution. We use the protocol's model to infer these values and needed additions. We then test the final execution against the protocol specification to ensure it is a valid protocol execution. Finally, we check if the resulting counterexample is indeed a violation of an isolation guarantee.

```
datatype txn_state =                    -- transaction state
  cl_prepared (ts, key ⇀ txid, key ⇀ value) |
  cl_committed (ts, key ⇀ txid, key ⇀ value) |
  ... -- other constructors: same as S2PL

record cl_conf =                        -- client configuration
  cl_state : txn_state
  cl_sn : nat
  cl_local_time : ts

datatype ver_state =                    -- version state
  prepared (ts, txid⊥, value⊥) |
  committed (ts, txid⊥, value⊥) |
  ... -- other constructors: same as S2PL

record svr_conf =                       -- server configuration
  svr_state : txid → ver_state

record global_conf =                    -- global configuration
  cls : cl_id → cl_conf
  svrs : key → svr_conf
  commit_order : key → list(txid)  -- a history variable
```

Figure 10: TAPIR client, server, and global configurations.

6.2 Case Study: TAPIR Protocol

TAPIR is a distributed transaction protocol, consisting of 2PC combined with optimistic concurrency control (OCC), which claims to offer *strictly serializable* (SSER) 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 offers fault tolerance while eliminating the coordination redundancy found in conventional replicated databases like Spanner [14].

We model TAPIR similar to S2PL, except that (i) TAPIR servers perform OCC checks instead of locking to prepare a read or write, (ii) operation results are stored directly in the version states instead of keeping version lists or key fingerprints, and (iii) the client's state contains a local clock for generating timestamps. These local clocks are *loosely synchronized*, and the authors state that the protocol only depends on their synchronization for performance but not for correctness. Figure 10 shows the TAPIR protocol's configurations.

Figure 11 presents our modeling of the tapir_occ_check function from [58] in VeriIso. The function's arguments are the key k, transaction ID t, the proposed timestamp ts, the read version's transaction ID t_r, and the written value v_w ($\perp$ for t_r or v_w indicates no reads or writes on key k). The check results in a new server state, either aborted or prepared. Note that TAPIR's OCC check returns ABSTAIN or RETRY in certain cases (see comments in Figure 11). These are used to improve performance by trying to avoid certain unnecessary aborts that would otherwise arise due to TAPIR's interaction with the underlying (inconsistent) replication protocol. As we do not model replication, we map both of these to aborted, which is safe in that it cannot introduce any new violations of SSER.

There is a different, earlier version of the OCC check [57], which corresponds to replacing one of the conditions for preparing new reads (Lines 14 and 15 in Figure 11) by the two lines in Figure 12. TAPIR's codebase [59] uses the newer definition in [58] (Figure 11).

```
definition prepared_rd_tstmps(k) ⟷
  {ts | svr_state(k, t) = prepared(ts, r, w) ∧ r ≠ ⊥}

definition prepared_wr_tstmps(k) ⟷
  {ts | svr_state(k, t) = prepared(ts, r, w) ∧ w ≠ ⊥}

definition committed_wr_tstmps(k) ⟷
  {ts | svr_state(k, t) = committed(ts, r, w) ∧ w ≠ ⊥}

definition tapir_occ_check(k, t, ts, t_r, v_w) ⟷
  if t_r ≠ ⊥ ∧ committed_wr_tstmps(k) ≠ {} ∧
     ver_ts(svr_state(k, t_r)) < Max(committed_wr_tstmps(k))
  then aborted
  else if t_r ≠ ⊥ ∧ prepared_wr_tstmps(k) ≠ {} ∧
          ts > Min(prepared_wr_tstmps(k))
  then aborted      -- originally: ABSTAIN
  else if v_w ≠ ⊥ ∧ prepared_rd_tstmps(k) ≠ {} ∧
          ts < Max(prepared_rd_tstmps(k))
  then aborted      -- originally: RETRY
  else if v_w ≠ ⊥ ∧ committed_wr_tstmps(k) ≠ {} ∧
          ts < Max(committed_wr_tstmps(k))
  then aborted      -- originally: RETRY
  else prepared(ts, t_r, v_w)
```

Figure 11: TAPIR OCC check, modeled after [58]'s definition.

```
  else if t_r ≠ ⊥ ∧ prepared_wr_tstmps(k) ≠ {} ∧
    ver_ts(svr_state(k, t_r)) < Min(prepared_wr_tstmps(k))
```

Figure 12: TAPIR OCC check's different condition for preparing reads, based on the (older) conference definition [57].

6.3 Finding TAPIR Isolation Bugs using VerIso

Lu et al. [39] have found an issue with SSER's real-time ordering in TAPIR but conjecture that it still satisfies SER. Using VerIso, we discover that TAPIR violates *atomic visibility* (RA), a much weaker isolation level than (S)SER. This bug escaped both the TAPIR authors' manual correctness proofs (including SSER) and their additional TLA+ model checking analysis. This underscores the need for our *rigorous* verification approach, which covers *all* system behaviors.

As TAPIR aims to provide SSER, we set up a corresponding refinement proof in VerIso. The event mapping π maps the client commit event to the abstract commit, and all other events to skip. The state mapping r reconstructs the abstract KVS's version lists by mapping the transaction IDs (recorded in `commit_order` history variable, in the order of client commits) to the corresponding abstract versions extracted from their version state. As for S2PL, the abstract view is the constant initial view.

For SSER, the view `u` used in the commit event must contain all abstract versions. Using general lemmas about full views, we prove the abstract guards for view wellformedness (`wf`), extension ($\mathcal{U}(cl) \sqsubseteq u$), and closure (`canCommit`). However, we could not prove the abstract LWW guard, despite following the steps in Section 4.3.2. As LWW must hold for any isolation level of our abstract model, this indicates that TAPIR potentially violates atomic visibility (RA). We confirm this by exhibiting two witnessing counterexamples, one for each version of the OCC check. These remain valid when TAPIR is layered on top of the replication protocol.

Example 5 (journal version [58]). We start from the transactions' interleaving shown in Figure 9a (top) and try assigning timestamps

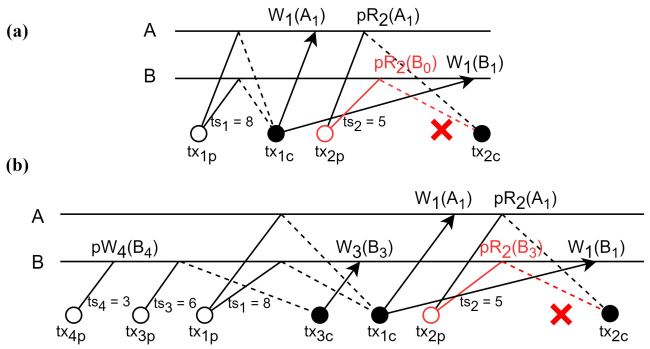

Figure 13: Atomic visibility counterexamples for TAPIR, based on `tapir_occ_check`'s definition in (a) [58] and (b) [57].

to each transaction such that they can prepare their reads and writes. The write transaction tx_1 can prepare its writes with any proposed timestamp $ts_1 > 0$ because initially there are no committed writes or prepared reads. Hence, the $\texttt{tapir\_occ\_check}(k, tx_1, ts_1, \bot, v)$ calls for $k \in \{A, B\}$ return `prepared`, as they are not aborted by any of the checks in Figure 11. The read transaction tx_2 calls $\texttt{tapir\_occ\_check}(k, tx_2, ts_2, t_r, \bot)$, for $k \in \{A, B\}$. Since at the execution time of these server prepare events only A_1 is committed, the maximum of committed writes' timestamps are ts_1 and 0 on A and B respectively, which are equal to the timestamps of the versions read by t_2, i.e., they are not aborted at Line 13. Furthermore, B has one prepared write (B_1), so $ts_2 \leq ts_1$ must hold to prevent $pR_2(B_0)$ from aborting on Line 16. Therefore, as shown in Figure 13a, by assigning $ts_1 = 8$ and $ts_2 = 5$, we get a valid execution of TAPIR that violates atomic visibility, as tx_1's writes are only partially observed by tx_2.

Example 6 (conference version [57]). We modify our previous counterexample to obtain a counterexample for the older definition (Figure 12), where the read version's timestamp (denoted by $\texttt{ver\_ts}(\texttt{svr\_state}(k, t_r))$) instead of the transaction's proposed timestamp is checked for preparing reads. In Figure 13a, at the time of tx_2's prepare read events, there is one prepared write on B with $ts_1 = 8$, which means the prepare read on B would abort on Line 16 ($0 < Min\{8\}$). So the read transaction should read a slightly newer version on B (see Figure 13b), for example, a single write by tx_3 on B with $ts_3 = 6$ (still older than $W_1(B_1)$), and also have another transaction tx_4 prepare a write on B with a lower timestamp, e.g., $ts_4 = 3$, so that it pulls the minimum down and $6 \not< Min\{3, 8\}$ prevents aborting on Line 16. We then check if the two new write transactions also pass the checks. As these transactions' prepare events appear before any prepare_read or commit_write events, the corresponding sets are empty, so both transactions successfully prepare their writes. Figure 13b shows the completed example, depicting a valid execution of TAPIR that violates atomic visibility, as tx_1's writes are only partially observed by tx_2.

Validation. We confirmed the violation of atomic visibility in TAPIR's implementation [59]. We treated TAPIR as a black box to preserve its codebase and minimize the risk of introducing errors, and collected transaction execution histories from its runs. For these runs, we employed TAPIR's own workload generator with a small configuration with 3 clients, 2 shards, 2 keys, 2 operations per transaction, and 50%

reads. We then analyzed the collected histories using IsoVista [21], a tool designed to detect isolation anomalies across various isolation levels. Its RA checking component identified violations of atomic visibility in these histories; see Appendix A for details.

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 introduced by Bernstein et al. [7] for characterizing serializability and later adapted by Adya to weaker isolation levels such as SI and RC [2].

Based on *abstract executions*, Cerone et al. [11] 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 [8] present an alternative axiomatic framework for characterizing isolation criteria via the write-read relation which, in contrast to the visibility relation [11], can be extracted straightforwardly from histories.

More recently, effective isolation checkers have been developed based on such declarative specifications [8, 21, 23, 28, 33, 54]. These include Biswas and Enea's dbcop [8], and Elle [28], based on dependency graphs [2], for checking multiple isolation levels, and Cobra [54] for SER and PolySI [23] for SI. All these tools verify whether individual database execution histories satisfy the required isolation level. Hence, they can only show the presence of errors, not their absence, in database implementations. Given that many isolation bugs stem from design-level defects in carefully designed and heavily tested production databases, full verification of database designs is highly desirable. Our VeriIso framework fulfills this requirement.

Operational Specifications. Crooks et al. [15] introduce a state-based formalization of isolation levels based on complete transaction traces; VeriIso only uses the current database state and clients' views on that state. Liu et al. [36] develop a formal framework in Maude [13] for specifying and model checking isolation properties of a database execution history. Lesani et al. [29] propose an operational framework for the verification of causally-consistent KVSs.

We base our framework VeriIso on the operational semantics of isolation guarantees in [56]. In contrast to the aforementioned related work, VeriIso supports *invariant-based verification*, which can be mechanized using theorem provers like Isabelle/HOL. Most related works [2, 7, 11, 15, 56] only provide theories of isolation criteria with no tool support for formal analysis. Similar to VeriIso, the framework in [29] is formalized in in a theorem prover, namely, Coq. However, it only supports non-transactional KVSs and causal consistency. The framework in [36] 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, VeriIso supports rigorous verification of arbitrary numbers of processes and transactions.

8 DISCUSSION AND CONCLUSION

We take a step back to review and discuss our approach. Table 2 shows some statistics of our formalization effort.

Table 2: Statistics of the Isabelle/HOL formalization.

	Tot LoC	Model LoC	Proof LoC	#lemmas	#inv.
VeriIso framew.	3093	–	–	275	11
S2PL+2PC	1890	285	1605	70	13
TAPIR	951	386	565	14	12
Total	5934	671	2170	359	36

Framework Infrastructure and Guarantees. VeriIso covers a wide spectrum of isolation levels ranging from RA to SSER and can thus be applied to a large variety of protocols. VeriIso provides both methodological guidance and structural support for protocol verification, including a protocol modeling style and a verification methodology and infrastructure. The latter consists of a sizable collection of supporting lemmas and proof rules (cf. first row of Table 2). Our correctness proofs provide strong guarantees, as they cover arbitrary numbers of processes and transactions, while model checking and testing can only find bugs, but not prove their absence.

Modeling and Verification Effort. Protocol modeling requires relatively little effort (e.g., a few days) and can be done by developers with some experience in functional programming. However, protocol verification using an interactive prover like Isabelle requires considerable expertise. The effort depends on the protocol and may range from a few weeks to several months.

Reusability. The abstract model can easily be instantiated to additional isolation levels stronger than RA, given a suitable dependency relation R_{IL} and session guarantee $\texttt{vShift}_{IL}$. For protocol modeling, we can reuse the top-level structure of configurations and the 2PC structure of events. For verification, we have identified several categories of invariants reusable or adaptable to other protocol verification efforts. Protocols designed for particular isolation levels come with their own types of invariants. For example, in [20], we have verified a TCC protocol, which required numerous invariants about timestamps. We expect these to be reusable for other TCC protocols.

Limitations. Our framework is based on the KVS model. It supports neither general relational databases nor predicate queries. Doing so would require a study of the formal semantics of weak isolation levels in such a setting. We also do not model any kind of failures. In particular, the 2PC protocol underlying most protocols is itself blocking. Some fault tolerance might be achievable using replication, which we consider to be orthogonal, as mentioned earlier.

Conclusions and Future Work. VeriIso is the first systematic, mechanized framework for formally verifying that database transaction protocols conform to their intended isolation levels. We demonstrated its effectiveness through two case studies, involving both verification and falsification. Given repeated occurrences of isolation bugs in database protocol designs, our work can help design more reliable transaction systems.

We plan to extend VeriIso with additional isolation guarantees, including weaker levels such as RC, and with additional infrastructure to facilitate refinement proofs. For example, we could refine our abstract transaction model into a generic abstract protocol model capturing the 2PC structure of many protocols, thus factoring out recurring parts of refinement proofs. This should facilitate the proof that concrete protocol models refine the abstract protocol model. Moreover, extending the correctness guarantees from designs to implementations would be desirable, e.g., using the approach in [52].

REFERENCES

[1] Martín Abadi and Leslie Lamport. 1991. The existence of refinement mappings. *Theoretical Computer Science* 82, 2 (1991), 253–284.

[2] 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.

[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] Azure. 2022. azure-cosmos-tla. https://github.com/Azure/azure-cosmos-tla.

[5] 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.

[6] 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.

[7] Philip A Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)* 13, 2 (1981), 185–221.

[8] Ranadeep Biswas and Constantin Enea. 2019. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 165:1–165:28.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[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] ElectricSQL. 2024. https://electric-sql.com/.

[20] Shabnam Ghasemirad, Christoph Sprenger, Si Liu, Luca Multazzu, and David Basin. 2024. *Pushing the Limit: Verified Performance-Optimal Causally-Consistent Database Transactions.* Technical Report. https://doi.org/10.48550/ARXIV.2411.07049 arXiv:2411.07049 [cs.DB]

[21] 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).

[22] Diana Hsieh. 2017. CockroachDB beta passes Jepsen testing. https://www.cockroachlabs.com/blog/cockroachdb-beta-passes-jepsen-testing/.

[23] 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.

[24] Jepsen. Accessed in May, 2024. Jepsen Analyses. https://jepsen.io/analyses.

[25] 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.

[26] Kyle Kingsbury. 2018. Dgraph 1.0.2. https://jepsen.io/analyses/dgraph-1-0-2.

[27] Kyle Kingsbury. 2019. YugaByte DB 1.3.1. https://jepsen.io/analyses/yugabyte-db-1.3.1,polysi.

[28] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (2020), 268–280.

[29] 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

[30] Galera Cluster Library. 2023. Remove SNAPSHOT ISOLATION mention from FAQ page #349. https://github.com/codership/documentation/commit/cc8d6125f1767493eb61e2cc82f5a365ecee6e7a.

[31] Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (1975), 717–721.

[32] 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).

[33] Si Liu, Long Gu, Hengfeng Wei, and David Basin. 2024. Plume: Efficient and Complete Black-box Checking of Weak Isolation Levels. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024).

[34] 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).

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] Nancy A. Lynch and Frits W. Vaandrager. 1995. Forward and Backward Simulations: I. Untimed Systems. *Inf. Comput.* 121, 2 (1995), 214–233.

[41] 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.

[42] Microsoft. 2024. Azure CosmosDB DB. https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels.

[43] MongoDB. 2024. https://www.mongodb.com/.

[44] Neo4j. 2024. https://neo4j.com/.

[45] 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

[46] Tobias Nipkow and Gerwin Klein. 2014. *Concrete semantics: with Isabelle/HOL*. Springer. https://doi.org/10.1007/978-3-319-10542-0

[47] 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

[48] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson (Eds.). 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-45949-9

[49] Christos H Papadimitriou. 1979. The serializability of concurrent database updates. *Journal of the ACM (JACM)* 26, 4 (1979), 631–653.

[50] 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.

[51] 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.

[52] 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

[53] Informal Systems. 2024. Apalache. https://apalache.informal.systems/.

[54] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. COBRA: Making Transactional Key-Value Stores Verifiably Serializable *(OSDI'20)*. USENIX Association, Article 4.

[55] 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.
[56] 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.
[57] 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.
[58] 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.
[59] Irene Zhang, Naveen Kr. Sharma, Michael Whittaker, and Maxime Caron. 2019. tapir. https://github.com/UWSysLab/tapir.

A VALIDATING RA VIOLATIONS IN TAPIR

Using the black-box database isolation checker IsoVista [21], we confirmed the violation of *atomic visibility* in TAPIR's implementation [59] that complies with the specification in its journal version [58]. By treating TAPIR as a black box, we preserved its codebase, thereby minimizing the risk of introducing errors on our end.

Validation Process. Our black-box validation of *read atomicity* (RA) violations in TAPIR is conducted using the following steps:

(1) TAPIR clients issue transactional requests generated by its workload generator;
(2) Each client session logs the requests it sends and the corresponding results returned by the database;
(3) The logs from all the client sessions are merged into a unified history;
(4) IsoVista then analyzes this history to determine whether it satisfies RA.

Specifically, in Step (4), IsoVista constructs a Biswas and Enea's dependency graph [8] from the history, and searches for specific subgraphs that represent various RA anomalies like *fractured reads* (see also Section 2.1).

Experimental Setup. We co-located the client threads (or sessions) in a local machine with an Intel Core i3-1005G1 CPU and 8GB memory. We set the parameters of TAPIR's workload generator as:[1]

```
nshard=2          # number of shards
nclient=3         # number of clients to run (per machine)
nkeys=2           # number of keys to use
rtime=1           # duration to run
tlen=2            # transaction length
wper=50           # writes percentage
err=0             # error
skew=0            # skew
zalpha=-1         # zipf alpha (-1 to enable uniform)
```

Counterexample. Figure 14 shows a counterexample of RA, specifically a fractured reads anomaly, that has been detected, minimized, and visualized by IsoVista. This counterexample involves three transactions from two clients: Txn(...67) and Txn(...68) from Clt(...70), along with Txn(...81) from Clt(...01).[2] It arises from conflicting dependencies between Txn(...67) and Txn(...68), represented by the cycle shown in Figure 14. This cycle is built as follows:

- Based on Biswas and Enea's theory [8], since (i) Txn(...81) reads Txn(...67)'s write to Key(...87), (ii) Txn(...81) reads Txn(...68)'s write to Key(...56) and (iii) Txn(...68) also writes to Key(...87), IsoVista *infers* a version order (or WW dependency) on Key(...87) from Txn(...68) to Txn(...67).
- Since Txn(...67) is issued before Txn(...68) in the same session, there is a session order (or SO dependency) from Txn(...67) to Txn(...68).

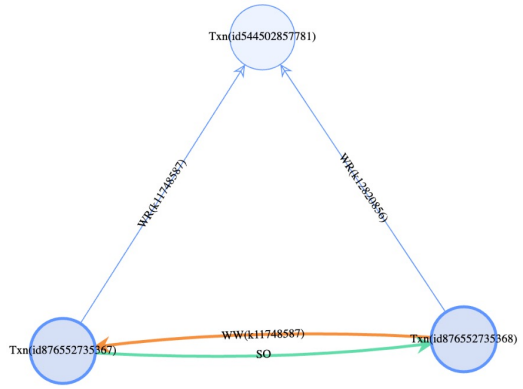

Figure 14: A counterexample of RA returned by IsoVista, which corresponds to the fracture reads anomaly.

Intuitively, this counterexample exhibits fractured reads because Txn(...81) observes only part of Txn(...68)'s updates. Specifically, Txn(...68)'s write to Key(...56) is visible to Txn(...81), while its write to Key(...87) is not; instead, Txn(...81) reads an older version of Key(...87) written by Txn(...67).

[1] Our fork of the TAPIR codebase, along with the experimental setup and result, is available at https://github.com/lucamul/TapirCorrectnessTest.
[2] The client IDs can be seen in IsoVista by expanding the nodes in Figure 14.