ETH zürich

Master Thesis

# Mechanized Data Consistency Models for Distributed Database Transactions

Shabnam Ghasemirad

Supervisors

Dr. Christoph Sprenger
Dr. Si Liu

Professor

Prof. David Basin

November, 2022

**Information Security Group**

**D-INFK – ETH Zürich**

**Abstract**

Cloud database systems use transactions for synchronization. Depending on the desired scalability and availability, different systems provide different transactional consistency guarantees, also known as data consistency models in the literature.

In this thesis, we have formalized an operational semantics framework for consistency models. This framework can be used to verify that concurrency control protocols implement their claimed notion of consistency and to analyze client programs' robustness under different consistency models. The focus of the thesis is on the former. To this end, we have modeled the Two-Phase Locking protocol (built on a Two-Phase Commit infrastructure system) in Isabelle/HOL and verified that it satisfies serializability. Furthermore, we have modeled and optimized the Eiger-PORT protocol, a state-of-the-art protocol satisfying Causal Consistency.

Finally, we will discuss some characteristics and implications of this framework and possible ways to extend or generalize it. Our long-term goal is to make the framework stronger and more generic to cover a wider range of protocols eligible for verification against consistency models.

**Keywords:** Consistency Models, Distributed Database Transactions, Isolation Levels, Verification Framework

# Acknowledgements

I am sincerely grateful for having the opportunity to conduct my master's thesis research in the Information Security group. It has been an invaluable experience, and I have learned a lot from my supervisors, Dr. Christoph Sprenger and Dr. Si Liu, and my professor, Prof. David Basin.

I would like to express my deepest gratitude to Dr. Christoph Sprenger for his support and guidance during this project, and for the long feedback sessions of problem solving and discussing ideas, where he generously provided me with his knowledge and expertise.

I would like to extend my sincere thanks to Dr. Si Liu for his guidance and valuable input during this project, and for his inspiring ideas for this thesis and the further development of our framework in the future.

Lastly, I would like to thank Johannes Dollinger for his love and emotional support during this process.

Finishing this project would not be possible without these people's help, support and encouragement.

# Contents

# Glossary and List of Acronyms

**RA** Read Atomic.

**CC** Causal Consistency.

**RYW** Read Your Writes.

**PSI** Parallel Snapshot Isolation.

**SI** Snapshot Isolation.

**SER** Serializability.

**KVS** Key-Value Store.

**SO** session order.

**WR** write-read.

**WW** write-write.

**RW** read-write.

**TM** Transaction Manager.

**RM** Resource Manager.

**KM** Key Manager.

**TID** Transaction Identifier.

**2PC** Two-Phase Commit.

**2PL** Two-Phase Locking.

**CC+** Causal+ Consistency.

**CCv** Causal Convergence.

**lst** local safe time.

**gst** global safe time.

# Chapter 1

# Introduction

Cloud database systems use transactions for synchronization. Depending on the desired scalability and availability, different systems provide different transactional consistency guarantees, also called data consistency models in the literature. Prevalent consistency models include Read Atomic (RA) [1], Causal Consistency (CC) [2], Parallel Snapshot Isolation (PSI) [3], Snapshot Isolation (SI) [4], and Serializability (SER) [5].

Different concurrency control mechanisms have been proposed that implement these consistency notions. For example, RAMP [1] satisfies RA, COPS [6] and Eiger-PORT [7] satisfy CC, and Two-Phase Locking implements SER. The complex concurrent behaviors of these protocols call for a formal verification that they satisfy the desired consistency model. Moreover, a formal analysis of transactional client programs is also desirable. Different types of formal semantics have been studied for these models.

*Declarative semantics* have been previously introduced for these models, using dependency graphs [8] and abstract executions [9]. This type of semantics is quite concise but also quite abstract. While it is possible to use abstract executions to prove that a concurrency control protocol is implementing its intended consistency model, the complete lack of state information seems to make this type of model unsuitable for proving the properties of client programs.

Xiong et al. in [10] define an *operational semantics* for representing different consistency models in a unified way. This model is centralized: the database is represented as a single, multi-versioned Key-Value Store (KVS). In reality, the database may be sharded and replicated, and each client may have a different view on its current content. These *client views* are explicitly represented in the centralized model as subsets of versions of each key that a given client sees.

Transactions are executed atomically, and the desired notion of consistency is specified as a so-called *execution test*, which consists of a condition that must be satisfied for a commit to take place and a constraint on the possible updates of the committing client's view on the database. The commit condition is formulated in terms of the write-read (WR) and write-write (WW) dependency relations and the read-write (RW) anti-dependency relation, first introduced in Adya's PhD thesis [8].

This model can be used both for verifying that concurrency control protocols implement their claimed notion of consistency and for analyzing client programs' robustness under different consistency models.

In this thesis, we have set up a mechanized framework in Isabelle/HOL for studying the correctness of concurrency control protocols and of client programs using databases based on these protocols. Concretely, we have

- formalized and mechanized the operational framework for consistency models proposed in [10], proved assumptions of this framework as invariants, and
- used it to study two concurrency control protocols, namely the Two-Phase Locking protocol satisfying SER, and the Eiger-PORT protocol [7], satisfying CC. We modeled the Two-Phase Locking protocol (built on a Two-Phase Commit infrastructure system) in Isabelle/HOL and verified that it satisfies serializability. We modeled and optimized the Eiger-PORT protocol, named Eiger-PORT+, to satisfy a stronger consistency model, namely Causal Convergence. The correctness proof of the second protocol is in progress.

These correctness proofs establish that at the time of committing a transaction, the protocol state can be related to an abstract state of the centralized model, which satisfies the execution test of the desired notion of consistency. We have paid particular attention to the proof techniques for establishing the correctness of such protocols. While the centralized model executes transactions atomically, the concrete protocol executes the constituting read and write operations one-by-one.

Two-Phase Locking (2PL) is a standard, widely used locking protocol in databases, and Eiger-PORT is a state-of-the-art representative implementation of its respective consistency model, CC, in the literature. Using these two protocols, we show the practicality of our framework and discuss the properties of such a framework and the functionalities it can provide.

To our knowledge, this is the first formalization of a general framework of consistency models and the first general and fully mechanized correctness proof of a concurrency control protocol in such a framework. The results of this study provide a foundation for further theoretical work, such as framework generalizations and modifications, and serve as the basis for further practical verification work, applicable to both protocols and client programs.

This report is organized into six chapters. Chapter 2 explains the fundamental concepts. Chapter 3 introduces the mechanized framework of data consistency models that we have developed in Isabelle/HOL and the properties of this framework. Chapters 4 and 5 contain two examples of protocol verification using the mechanized framework and the proof techniques we have used. Chapter 6 discusses related work, including abstract executions and dependency graphs approaches, and Chapter 7 concludes the thesis and explores possible directions for future work.

# Chapter 2

# Preliminaries

In this chapter, we introduce the fundamental concepts used in the thesis. In Section 2.1, we discuss distributed systems and related topics, such as dependency relations and logical clocks. Isabelle/HOL, which we use to formalize our development, is briefly introduced in Section 2.2. Finally, following the presentation of [11], we explain event systems, invariants, and refinement in Section 2.3.

Due to the fact that the operational framework for consistency models introduced in Xiong et al. [10] and our mechanized framework are closely related and inseparable, we present the definitions of this framework together with our formalization in the next chapter.

## 2.1 Distributed Database Systems

Distributed systems are prevalent in today's computing infrastructures. A distributed system consists of independent components that are spread across different machines or locations. These components send messages to each other to synchronize and work together for a common purpose. Due to the growing demand for more data to be stored in databases, distributed database systems are becoming increasingly popular. One of the most critical aspects of these systems is the level of consistency and availability they can provide.

According to the CAP theorem [12], a distributed database system can only have two out of three following properties: (strong) Consistency, Availability, and Partition tolerance. Most distributed database systems need to prioritize availability and partition tolerance to provide a reliable service to their customers. As a result, they need to settle for weaker notions of consistency than serializability.

A wide range of data consistency models have been proposed and defined to meet the requirements of different systems. The choice of consistency level is always a trade-off, as the stronger or more strict the consistency model, the lower the availability [13]. Since different distributed database protocols satisfy different data consistency models, depending on the system's need, a suitable protocol can be chosen or even created from scratch for that system.

This is where the verification of distributed database protocols becomes relevant; So that we can rely on a protocol designed to satisfy a certain consistency model. There are some well-known protocols in the literature, such as two-phase locking, which have been proven to satisfy a particular consistency model, in this case, serializability. However, in recent years, there has been a growth in the number of designed protocols that have better performance or are more memory or communication efficient but might only satisfy weaker consistency models. An effective verification method is necessary to ensure the reliability of these state-of-the-art protocols and guarantee the consistency level they provide.

In the following sections, we will review the definition of data consistency models and read and write dependency relations that form the basis of formal definitions for consistency models. Finally, we will briefly introduce logical clocks, used in many distributed database protocols for capturing chronological and causal relationships.

### 2.1.1 Data Consistency Models and Isolation Levels

Distributed database systems use transactions for synchronization. Depending on the desired scalability and availability, different systems provide different transactional consistency guarantees, also called data consistency models in the literature.

The first consistency models were initially defined by engineers and were formulated quite informally for an implementation purpose or specified by the forbidden behaviors or anomalies.

One of the first and most famous classifications for consistency levels is the ANSI isolation levels [14]. It comprises the four following levels of SQL isolation:

1. *read uncommitted*: In this model, a transaction can read uncommitted values.
2. *read committed*: The transactions only read committed values, but the same read in a transaction might return two different values.
3. *repeatable read*: Repeating the same read always returns the same committed value for a transaction.
4. *serializability*: Executing the transactions has the same effect as some sequential execution of those same transactions.

Since these informal definitions were ambiguous in some cases, formal definitions of these were proposed, and new consistency models were formulated over the years. We will briefly explain some of the most important consistency models.

**Read Atomicity (RA).** In this model, if a transaction $t$ reads a version of a key written by a transaction $t'$, then any other versions written by $t'$ for other keys are the oldest versions that $t$ might read for those keys. In other words, $t$ will either read those versions or newer versions that override them if available.

**Causal Consistency (CC).** In this model, the causal relationship of the transactions is preserved. For example, if a transaction $t$ observes the effects of a transaction $t'$, it should also be able to observe all the effects that are visible to $t'$.

**Parallel Snapshot Isolation (PSI).** This model has the same guarantees as CC, and additionally requires that there are no write conflicts.

**(Strict) Serializability ((S)SER).** This is the strongest consistency guarantee and requires that there exists a total sequential order on transactions and their visible effects on other transactions.

Figure 2.1 gives an overview of the relations between different known consistency models. The arrows go from weaker to stronger consistency models. In other words, if there is an arrow from a to b, b guarantees more consistency conditions while satisfying all the conditions required by a. The consistency models covered by our framework are highlighted with a blue color.

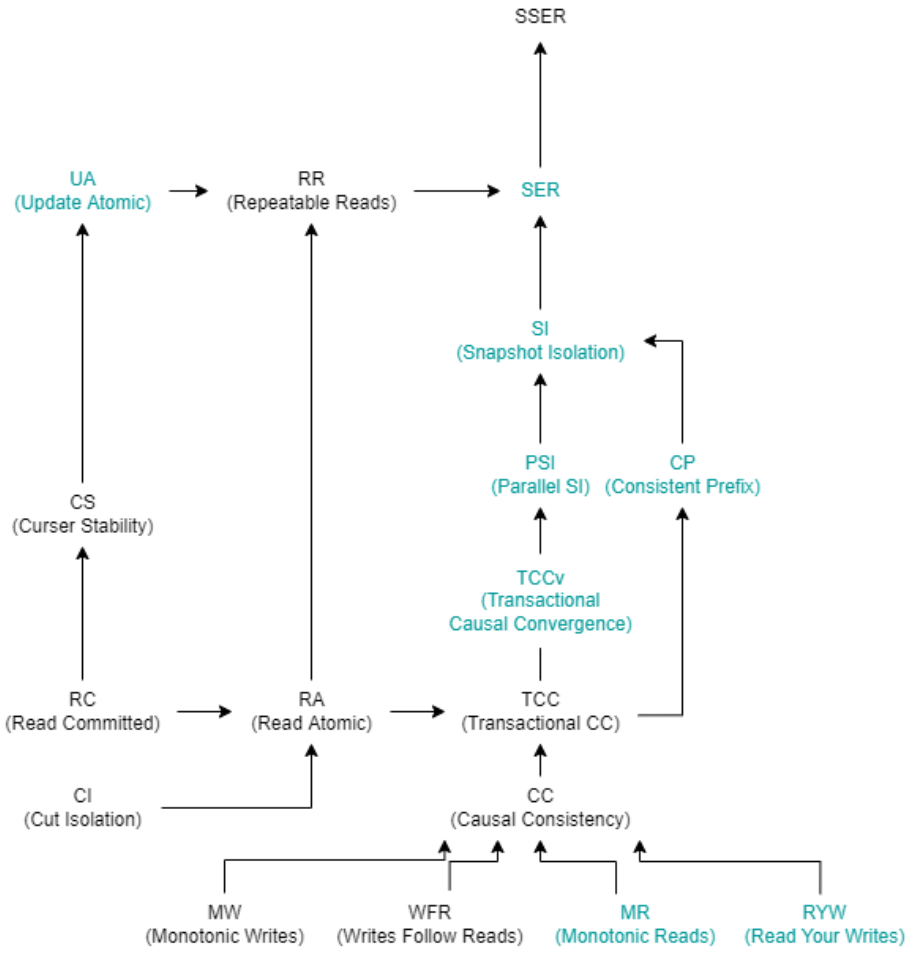


Figure 2.1: A hierarchy of data consistency models

The relation between consistency models is not always obvious from their formal definitions. One of the numerous benefits of mechanizing the consistency model framework is that we can

use it to show implications between these models similar to what the diagram of Figure 2.1 illustrates. Other benefits, such as certified protocol verification of data consistency, have been discussed in more detail in the rest of the thesis.

### 2.1.2 Dependency Relations

Dependency relations for distributed database transactions have been proposed by Adya's PhD thesis [8] and used for dependency graphs. These dependency relations are a basis for formalizing consistency models. There are three possible dependency relations between two transactions, namely *write-read* (WR), *write-write* (WW), and *read-write* (RW). The formal definitions of these dependency relations, adapted to key-value stores, are presented below, following the representation of Xiong's PhD thesis [15].

**Definition 1** (WR)**.** write-read dependency for kv-store $\mathcal{K}$ on key $k$, is defined as:

$$\mathbf{WR}_{\mathcal{K}}(k) \stackrel{\text{def}}{=} \{(t, t') \mid \exists i.\ t = v\_writer(\mathcal{K}(k, i)) \ \wedge \ t' \in v\_readerset(\mathcal{K}(k, i))\}$$

This means there is a write-read dependency on key $k$ between two transactions $t$ and $t'$, if $t'$ reads a version of $k$ written by $t$.

**Definition 2** (WW)**.** write-write dependency for kv-store $\mathcal{K}$ on key $k$, is defined as:

$$\mathbf{WW}_{\mathcal{K}}(k) \stackrel{\text{def}}{=} \{(t, t') \mid \exists i,\ i'.\ t = v\_writer(\mathcal{K}(k, i)) \ \wedge \ t' = v\_writer(\mathcal{K}(k, i')) \ \wedge \ i < i'\}$$

This means there is a write-write dependency on key $k$ between two transactions $t$ and $t'$, if they both have written a version of $k$ and $t'$ has written the newer version, i.e., it has overwritten the version written by $t$.

**Definition 3** (RW)**.** read-write anti-dependency for kv-store $\mathcal{K}$ on key $k$, is defined as:

$$\mathbf{RW}_{\mathcal{K}}(k) \stackrel{\text{def}}{=} \{(t, t') \mid \exists i,\ i'.\ t \in v\_readerset(\mathcal{K}(k, i)) \wedge t' = v\_writer(\mathcal{K}(k, i')) \wedge i < i' \wedge t \neq t'\}$$

This means there is a read-write anti-dependency relation on key $k$ between two transactions $t$ and $t'$, if they are two different transactions where $t'$ writes a version of $k$, but $t$ reads an older version of $k$. In other words, $t'$ has committed after $t$ had finished its read.

### 2.1.3 Logical Clocks

A logical clock is a mechanism for capturing chronological and causal relationships in a distributed system. Most of the time, there is no global clock that all components of the distributed system can synchronize with. However, in many applications, it is enough for different components (processes) to agree on the order of events rather than an actual clock time [16].

There have been several algorithms designed based on logical clocks, such as Lamport timestamps [17], vector clocks [18], and version clocks [7]. The basic idea behind a logical clock is storing a local time and sending it with each message that exchanges data and communicates with other processes, and then updating the time accordingly, based on local events and received messages.

We consider Lamport clocks as an example to demonstrate how the algorithm works in practice and the guarantees that using a logical clock provides. This clock is also used in the Eiger-PORT protocol [7], discussed in Chapter 5.

The Lamport timestamp algorithm follows the following three rules:

1. Each process increments its clock before a local event or sending a message.

2. If a process is sending a message, it always includes the clock value after incrementation (rule 1) in the message.

3. Upon receiving a message in a process, the process updates its clock to the maximum value of its own clock and the timestamp stored in the message and then increments this value by one, which marks the end of the message-receiving event.

Lamport clocks create a partial ordering of events across different processes. If two events $a$ and $b$ have time stamps $t_a$ and $t_b$, and $a \rightarrow b$ holds, meaning $a$ happens before $b$ or influences $b$, necessarily by communicating, then $t_b$ is larger than $t_a$. On the other hand, two events may be incomparable because they had no interaction or effect on each other. In this case, their order is irrelevant, and they are assumed to happen *in parallel*. As a result, $a \rightarrow b$ implies $t_a < t_b$, known as the *clock consistency condition*, but the other direction does not necessarily hold; $t_a < t_b$ implies that either $a$ influenced $b$ or that they are incomparable, although we can be sure that $b$ has not influenced $a$ by contrapositive.

## 2.2 Isabelle/HOL

We have used the Isabelle/HOL proof assistant for our modeling and proofs. Isabelle is a generic system for implementing logical formalisms, and Isabelle/HOL is the specialization of Isabelle for Higher-Order Logic, which abbreviates to HOL [19] [20].

## 2.3 Event systems, refinement, and invariants

In this section, we introduce some important concepts of the Igloo framework [11], used for refinement proofs in this thesis, following the presentation of [11]. For details on notation and other definitions, see [11].

## Event Systems

Event systems are used to describe systems that work based on a series of events or labeled state transitions. Each event system consists of a set of states $S$, a set of events $E$, and a transition relation $\rightarrow \subseteq S \times E \times S$. We denote this for an event system $\mathcal{E}$ as $\mathcal{E} = (S, E, \rightarrow)$. We usually write $s \xrightarrow{e} s'$ for $(s, e, s') \in \rightarrow$.

The transitions relation can be extended to finite sequences of events $\tau$ by inductively defining, for all $s, s', s" \in S$, (i) $s \xrightarrow{\epsilon} s$ and (ii) $s \xrightarrow{\tau.<e>} s''$, whenever $s \xrightarrow{\tau} s'$ and $s' \xrightarrow{e} s''$.

A trace of an event system $\mathcal{E}$, starting in a state chosen from a set of initial states $I \subseteq S$, is a finite sequence $\tau$ of events such that $s \xrightarrow{\tau} s'$ for some initial state $s \in I$ and reachable state s'. We denote the set of all traces of $\mathcal{E}$ starting in I as $traces(\mathcal{E}, I)$. For singleton sets $I = \{s\}$, we also write $traces(\mathcal{E}, s)$, omitting the set brackets. We call a set of traces $P$ over $E$ a trace property and write $\mathcal{E}, I \models P$ if $traces(\mathcal{E}, I) \subseteq P$.

For concrete specifications, we often use a more structured form of event systems, called guarded event systems, of the form $\mathcal{G} = (S,\ E,\ G,\ U)$, where $S$ and $E$ are as for event systems, $G$ is the event indexed ($E$-indexed) families of guards $G_e : S \rightarrow \mathbb{B}$ and $U$ is the event indexed families of update functions $U_e : S \rightarrow S$. The associated transition relation is $\rightarrow = \{(s, e, s') \mid G_e(s) \wedge s' = U_e(s)\}$. If $S = (\!|\ \overline{x} \in \overline{T}\ |\!)$ is a record type, we use the notation $e : G_e(\overline{x}) \triangleright \overline{x} := U_e(\overline{x})$ to define events.

## Refinement

Refinement is defined between two event systems that usually represent different levels of abstractions, where the concrete one refines the more abstract one. For example, for $\mathcal{E}_1 = (S_1, E_1, \rightarrow_1)$ and $\mathcal{E}_2 = (S_2, E_2, \rightarrow_2)$ and sets of initial states $I_1 \subseteq S_1$ and $I_2 \subseteq S_2$, if we want $(\mathcal{E}_2, I_2)$ to refine $(\mathcal{E}_1, I_1)$, modulo a mediator function $\pi : E_2 \rightarrow E_1$, written $(\mathcal{E}_2, I_2) \sqsubseteq_\pi (\mathcal{E}_1, I_1)$, we must define a simulation relation $R \subseteq S_2 \times S_1$ that fulfills the following conditions:

1. for each $s_2 \in I_2$ there exists some $s_1 \in I_1$ such that $(s_2, s_1) \in R$.

2. for all $s_1 \in S_1$, $s_2, s_2' \in S_2$ and $e_2 \in E_2$ such that $(s_2, s_1) \in R$ and $s_2 \xrightarrow{e_2}_2 s_2'$ there exists some $s_1' \in S_1$ such that $s_1 \xrightarrow{\pi(e_2)}_1 s_1'$ and $(s_2', s_1') \in R$.

Refinement guarantees trace inclusion. A property is denoted by a set of traces, so trace inclusion guarantees property preservation. As a result, if we prove that certain properties hold for an abstract model and a concrete model refines the abstract model, then those properties will also hold for the concrete model.

### Invariant

An invariant is a property that remains unchanged under certain classes of transformations. In event systems, these transformations are the events, i.e., the state transitions. Finding invariants helps us understand and verify the system we are dealing with.

Invariants are extremely useful for classifying mathematical objects because they usually reflect the intrinsic properties of the object of study [21]. They can explain the model by classifying the different behavior of the system in different conditions and help with the refinements. The state invariants are proven by showing that they hold in the initial state and any state reachable by the events of the event system.

# Chapter 3

# Mechanized framework of data consistency models

We have formalized the operational framework for consistency models from [10] in Isabelle/HOL. As a part of this process, we have adapted the framework to Isabelle's mechanisms and limitations and added definitions as needed. Necessarily, this chapter includes many definitions from Xiong et al.'s work [10]. We have included remarks or explanations in the case of definitions that deviate from the original in our model. Moreover, we have proven some assumptions of the operational framework of [10] as invariants of our formalization, thereby strengthening the framework by eliminating the need to assume these conditions.

We divided the main framework into the following modules: First, a core execution test model as the abstract model, which is parametric and can generate different consistency models for different instantiations. This is the model that is later used for the verification of protocols by refinement. Second, a formalized programming language model, which is a refinement of the execution test model, and deals with client-local program stacks and snapshots, the necessary components for client programs' verification. We prove that both models preserve the well-formedness of key-value stores.

In this thesis, we mainly focus on the abstract *execution test* model for the verification of protocols, and we model two protocols for this purpose. We do not get into the details of how the client program handles the protocol execution. Instead, we are focused on the protocol's properties and how it satisfies a consistency model's requirements, i.e., the execution test.

Our framework, developed in Isabelle/HOL, provides a solid foundation for further practical verification work for protocols and client programs, and simplifies individual proofs. This is demonstrated by protocol verifications in the next two chapters. Client program properties can be proven similarly using the programming language module of our framework. Furthermore, property preservation through strengthening consistency models, using our framework, or generalizing the framework to support a broader range of consistency protocols are all possible directions for further theoretical research based on this mechanized framework of consistency models.

In the following sections, we discuss the operational framework of [10] and expand on our mechanized framework design, first introducing the abstract model, *execution test*, followed by the *programming language* model, which we prove refines the execution test model. At the end of the chapter, we explain how our framework is used for protocol verification using refinement proofs.

**Notation.** Through the rest of this chapter we use the following notation. For a given relation $r$, the reflexive, transitive, and reflexive-transitive closures of r are denoted by $r^?$, $r^+$, and $r^*$, respectively. The inverse of r is written as $r^{-1}$. For two relations $r$ and $r'$, the composition of relations $r$; $r'$ is equivalent to $\{(a_1, a_2) \mid \exists a.\ (a_1, a) \in r\ \wedge\ (a, a_2) \in r'\}$.

## 3.1 Key-value stores and views

The operational framework of Xiong et al. [10] introduces interleaving operational semantics for atomic transactions on a global centralized multi-versioned key-value store and client views.

**Definition 4** (Key-value store)**.** $\mathcal{K}$: KEY $\rightarrow$ List (VERSION). A key-value store is a function from keys to lists of versions. A version is modeled as a record type and consists of a value (v_value), a writer transaction (v_writer), and a set of reader transactions (v_readerset). Each version list is initialized with a version $v_0$ written by the $T0$ transaction.

**Definition 5** (Client view)**.** $u \in$ VIEWS($\mathcal{K}$) $\triangleq$ KEY $\rightarrow \mathcal{P}$ ($\mathbb{N}$). A function from keys to sets of natural numbers (version identifiers) representing the versions of each key that are visible to the client. The client view is always initialized to include the initial versions of each key; so for the initial view, $u_0$, the following holds: $\forall k \in KEY.\ u_0(k) = \{0\}$. An order between two views $u$ and $u'$ is defined by $u \sqsubseteq u'$, read as $u$ can grow/advance to $u'$ and holds when $\forall k.\ u(k) \subseteq u'(k)$.

**Definition 6** (Configuration)**.** A configuration is a pair ($\mathcal{K}$, $\mathcal{U}$) where $\mathcal{K}$ is a key-value store and $\mathcal{U}$ is a function from client identifiers to their view.

**Definition 7** (Snapshot)**.** $\sigma \in$ SNAPSHOT $\triangleq$ KEY $\rightarrow$ VALUE. A *transactional snapshot* is a function from keys to values. It usually stores key-value mappings before a transaction commit.

A *view snapshot* on a KVS, given a client view $u$, is a function from each key to the value of the newest version (with maximum version index) available in the view for that key, i.e., snapshot($\mathcal{K}$, $u$) $\triangleq \lambda k.$ v_value ($\mathcal{K}(k, max_<(u(k)))$).

**Definition 8** (Fingerprint)**.** A fingerprint is a set of operations in the form ($l$, $k$, $v$), where $l \in \{R, W\}$ is the operation mode, read or write, and $k$ and $v$ are the key and its corresponding value which is read or written. A fingerprint stores the reads and writes of a transaction to the keys. It only stores the *first read* and the *last write* on a key $k$ and ignores multiple reads and writes, or reads that come after a write to the same key.

**Remark:** A fingerprint contains at most one read and at most one write for each given key. Thus, in our modeling, we have defined a fingerprint as a partial function of keys to $(l, v)$ pairs, and we can divide the fingerprint into many *key* fingerprints, each for a given key.

**Definition 9** (Transaction Identifier (TID))**.** Transactions are identified using two constructors:

1. *T0*: The initial writer transaction that writes the initial version for all keys in the KVS. It does not belong to a specific client and has no sequence number.
2. *Tn (Tn_cl sn cl)*: Where the *cl* is the client who owns the transaction and *sn* is the sequence number which, together with the *cl*, makes the TID unique.

**Definition 10** (Session order)**.** The session order (SO), is defined by:

$$\mathbf{SO} \stackrel{\text{def}}{=} \{(t^n_{cl}, t^{n'}_{cl}) \mid t^n_{cl}, t^{n'}_{cl} \in \text{TXID} \ \wedge \ n < n'\}$$

Where *cl* is a client identifier, TXID is the set of transactions, and $t^n_{cl}$ denotes a transaction owned by client *cl* with a sequence number $n$.

**Remark:** The original definition, given in [15], has an extra condition $cl \in \text{CID}$, which we dropped since a client identifier can have any value of a fixed type $cl_id$, which we have declared using ***typedecl*** *cl_id* in Isabelle.

## 3.2 Limitations and assumptions

A ***last-write-wins*** policy is hard-wired in this framework. This means that when a client with several versions of a key in its view reads that key, it gets the value of the last version (maximum version id). So the atomic snapshot that the read transaction depends on, has the value of the last available updates for each key according to the client's view (see Definition 7). This poses some restrictions on the framework's compatibility with more complicated protocols, such as RAMP-1PW [1], where the client might need to read stale data sometimes.

The framework also makes four assumptions for the well-formedness of key-value stores and two for the well-formedness of client views.

**Well-formed KVS properties:**

1. snapshot: Each transaction present in the KVS, has read at most one version and written at most one version of each key.
2. ww_so: If there is a WW dependency between two transactions $t$ and $t'$ in the KVS ($t'$ writes a newer version), $t'$ can not appear before $t$ in $\text{SO}^?$.
3. wr_so: If there is a WR dependency between two transactions $t$ and $t'$ in the KVS ($t'$ reads a version written by $t$), $t'$ can not appear before $t$ in $\text{SO}^?$.
4. kvs_initialized: All keys in the KVS have at least an initial version $v_0$ in their version list.

**Remark:** The last property is implicitly assumed and is not explicitly mentioned for well-formedness of KVS, but we formalize it as one of the requirements for KVS well-formedness. We have proven these four properties as invariants for reachable states of both the execution test and the programming language models. More details of these proofs are provided in the next sections.

**Well-formed client view properties:**

1. in-range: For each key, the view always includes the initial version $v_0$. Additionally, all version identifiers are in the range of the key-value store, i.e., smaller than the corresponding version list's length.
2. atomic: All versions written by one transaction for different keys are either entirely included in the view or completely absent from it.

The **atomic view** combined with the ***last-write-wins*** policy provides an *atomic visibility* guarantee. However, the atomic view assumption is stronger than necessary because it requires all versions of all keys written by one transaction to be included, even if newer versions are available for some keys and the older ones are not read by any transactions anymore. This is analyzed in more detail in Chapter 5.

## 3.3 Execution Tests

In this section, we first introduce execution tests, which define the conditions that need to *locally* hold for a client to be allowed to commit an update to the KVS, and then a *global* transition rule for the abstract system model, over configurations (as abstract model's states).

**Definition 11** (Execution test)**.**

$$\frac{\textbf{canCommit}_{\textbf{ET}}\ (\mathcal{K},\ u,\ \mathcal{F}) \quad \textbf{vShift}_{\textbf{ET}}\ (\mathcal{K},\ u,\ \mathcal{K}',\ u') \quad \text{well-formed}(\mathcal{K}, u) \quad \text{well-formed}(\mathcal{K}', u') \quad \forall k, v.\ (R,\ k,\ v) \in \mathcal{F} \rightarrow \mathcal{K}(k,\ max_<(u(k))) = v}{cl \vdash (\mathcal{K},\ u) \xrightarrow{\mathcal{F}}_{ET} (\mathcal{K}',\ u')}$$

An execution test, **ET**, is a set of tuples of the form $(\mathcal{K},\ u,\ \mathcal{F},\ \mathcal{K}',\ u')$, with $\mathcal{K}$ and $\mathcal{K}'$ as KVSs, $u$ and $u'$ as client views, and $\mathcal{F}$ as a fingerprint, where the following conditions hold:

1. $u$ and $u'$ are well-formed;
2. $\textbf{canCommit}_{\textbf{ET}}\ (\mathcal{K},\ u,\ \mathcal{F})$;
3. $\textbf{vShift}_{\textbf{ET}}\ (\mathcal{K},\ u,\ \mathcal{K}',\ u')$;
4. $\forall k, v.\ (R,\ k,\ v) \in \mathcal{F} \rightarrow \mathcal{K}(k,\ max_<(u(k))) = v$. This is the *last_write_wins* policy for the fingerprint. We refer to this as the $fingerprint\_property$ from now on.

The notation $(\mathcal{K},\ u) \xrightarrow{\mathcal{F}}_{ET} (\mathcal{K}',\ u')$ is used in place of $(\mathcal{K},\ u,\ \mathcal{F},\ \mathcal{K}',\ u') \in ET$, to show that this rule can be thought of as a local transition on the KVS and the client $cl$'s view. The $\mathbf{canCommit_{ET}}$ and $\mathbf{vShift_{ET}}$ premises, parameterized by ET, are defined below.

**Definition 12** ($\text{canCommit}_{\text{ET}}$)**.** $\mathbf{canCommit_{ET}}(\mathcal{K}, u, \mathcal{F}) \triangleq \mathbf{closed}(\mathcal{K}, u, R_{ET})$.
This predicate indicates the conditions that are checked upon committing. It is parameterized by ET because these conditions vary depending on the consistency model. The $\text{canCommit}_{\text{ET}}$ predicate is defined as a *view closure* condition for $u$ with respect to the key-value store $\mathcal{K}$, and a relation $R_{ET}$. The relation $R_{ET}$ is parametric in ET and is specific to the consistency model. It is calculated with $\mathcal{K}$ and the transaction fingerprint $\mathcal{F}$ as inputs (see Table 3.1 for examples).

**View closure.** If the set of visible transactions for a client with the client view $u$ in the key-value store $\mathcal{K}$ is defined as $visTx(\mathcal{K}, u) \triangleq \{v\_writer(\mathcal{K}(k, i)) \mid i \in u(k)\}$, then the view $u$ is *closed* with respect to $\mathcal{K}$ and $R_{ET}$, written as $\mathbf{closed}(\mathcal{K}, u, R_{ET})$, if and only if:

$$visTx(\mathcal{K}, u) = ((R^*)^{-1}(visTx(\mathcal{K}, u))) \setminus \{t \mid \forall k \in \mathcal{K}, i.\ t \neq v\_writer(\mathcal{K}(k, i))\}$$

So if a transaction $t$ is visible in $u$, then all transactions $t'$ that are $R^*$-before $t$ ($(t', t) \in R^*$) and are not read-only are also visible in $u$.

**Definition 13** ($\text{vShift}_{\text{ET}}$)**.** This predicate indicates the conditions that are checked for advancing the client view in a commit. It is parameterized by ET, because these conditions vary depending on the consistency model. Under each consistency model, certain view shifts are not allowed, so a transaction can only commit a transaction when the updated view satisfies the $\text{vShift}_{\text{ET}}$ predicate, which is instantiated for the desired consistency model.

Table 3.1 is taken from Figure 6 in [10], and it shows the instances of $R_{ET}$ and $vShift_{ET}$ for execution tests of different consistency models. The dependency relations $WR_{\mathcal{K}}, WW_{\mathcal{K}}, RW_{\mathcal{K}}$ for a given key $k$ are defined in Section 2.1. The notation used here, without the input key, is calculated by taking the union over all keys.

| ET | $\mathbf{canCommit_{ET}}(\mathcal{K}, u, \mathcal{F}) \triangleq \mathbf{closed}(\mathcal{K}, u, R_{ET})$ | $\mathbf{vShift_{ET}}(\mathcal{K}, u, \mathcal{K}', u')$ |
|---|---|---|
| MR | **true** | $u \sqsubseteq u'$ |
| RYW | **true** | $\forall t \in \mathcal{K}' \setminus \mathcal{K}.\ \forall k,\ i.$ $(v\_writer(\mathcal{K}'(k,\ i)),\ t) \in SO^?$ $\Rightarrow\ i \in u'(k)$ |
| CC | $R_{CC} \triangleq SO \cup WR_{\mathcal{K}}$ | $\mathbf{vShift_{MR \cap RYW}}(\mathcal{K}, u, \mathcal{K}', u')$ |
| UA | $R_{UA} \triangleq \bigcup_{(W,k,\_))\in\mathcal{F}} WW_{\mathcal{K}}^{-1}(k)$ | **true** |
| PSI | $R_{PSI} \triangleq R_{UA} \cup R_{CC} \cup WW_{\mathcal{K}}$ | $\mathbf{vShift_{MR \cap RYW}}(\mathcal{K}, u, \mathcal{K}', u')$ |
| CP | $R_{CP} \triangleq SO;\ RW_{\mathcal{K}}^? \cup WR_{\mathcal{K}};\ RW_{\mathcal{K}}^? \cup WW_{\mathcal{K}}$ | $\mathbf{vShift_{MR \cap RYW}}(\mathcal{K}, u, \mathcal{K}', u')$ |
| SI | $R_{SI} \triangleq R_{UA} \cup R_{CP} \cup (WW_{\mathcal{K}};\ RW_{\mathcal{K}})$ | $\mathbf{vShift_{MR \cap RYW}}(\mathcal{K}, u, \mathcal{K}', u')$ |
| SER | $R_{SER} \triangleq WW_{\mathcal{K}}^{-1}$ | **true** |

Table 3.1: Instances of $\mathbf{R_{ET}}$ and $\mathbf{vShift_{ET}}$ for execution tests of consistency models.

**Definition 14** (Execution test global transition rule)**.**

$$\frac{\mathcal{U}(cl) \sqsubseteq u'' \quad cl \vdash (\mathcal{K},\ u'') \xrightarrow{\mathcal{F}}_{ET} (\mathcal{K}',\ u') \quad t \in \text{NextTxID}\ (cl,\ \mathcal{K}) \quad \mathcal{K}' = \text{UpdateKV}\ (\mathcal{K},\ u'',\ \mathcal{F},\ t)}{(\mathcal{K},\ \mathcal{U}) \xrightarrow{(cl,\ u'',\ \mathcal{F})}_{ET} (\mathcal{K}',\ \mathcal{U}[cl \mapsto u'])}$$

This rule is the global transition rule for the abstract model of execution tests, which uses the *local* transition rule of execution test (Definition 11) as a premise. This rule was originally defined as two separate rules for $(i)$ initial view shifting for synchronization $(ii)$ committing the update, according to Definition 4.26 of [15]. However, we combine these two rules and obtain the given rule above, which we later reuse in the definition and well-formedness proof of the programming language model.

In the framework we have developed, an execution test is modeled as a transition (event) system. The transitions or events of this system are the updates to the database that a particular data consistency model permits, i.e., the execution test global transition rule. The states of this transition system represent the valid or "consistent" states with regard to the data consistency model. So the *execution test* is a parameterized definition that depends on the desired data consistency model.

In Isabelle, we have modeled this using a *locale* and the arguments $R_{ET}$ and $vShift_{ET}$. Different consistency models can then be instantiated from the parameterized execution test (generic locale) to get a concrete transition system. We have added several widely used consistency models to the framework, but other models can also be added on demand simply by instantiating the execution test with the appropriate values for $R_{ET}$ and $vShift_{ET}$.

## Auxiliary definitions and Simplifications

Our mechanized framework introduces numerous lemmas and simplifications for different definitions used in the model, which makes the further use of the framework for verifying protocols much easier.

There are different functions defined for extracting transactions by their different features. For example, *NextTxID*, which is referred to in the execution test global transition rule (Definition 14), is implemented using:

- *vl_writers* a function for extracting all the transactions that have read from a specific key's version list by that point.
- *vl_readers* a function for extracting all the transactions that have written to a specific key's version list.
- *get_sqns* a function that combines the sequence numbers of the transactions calculated from the two previous functions.

- *next_txids*, which returns a set of eligible fresh transaction identifiers which can be used for choosing the next TID.

These are then followed by multiple lemmas about their properties that make using them in a verification proof easier.

As part of our mechanized framework, we have also introduced and formalized ordering relations among versions, version lists, and key-value stores. The definitions of those follow.

**Definition 15** (Version order)**.** A version order $v1 \sqsubseteq_{ver} v2$ is defined for two versions $v1$ and $v2$, and it holds if and only if all the following conditions hold:

$$\boldsymbol{v\_value}\ v1 = \boldsymbol{v\_value}\ v2$$
$$\boldsymbol{v\_writer}\ v1 = \boldsymbol{v\_writer}\ v2$$
$$\boldsymbol{v\_readerset}\ v1 \subseteq \boldsymbol{v\_readerset}\ v2$$

**Definition 16** (Version list order)**.** A version list order $vl_1 \sqsubseteq_{vl} vl_2$ is defined for two version lists $vl_1$ and $vl_2$, and it holds if and only if all the following conditions hold:

$$\boldsymbol{length}\ vl_1 \leq \boldsymbol{length}\ vl_2$$
$$\forall i < \boldsymbol{length}\ vl_1.\ vl_1[i] \sqsubseteq_{ver} vl_2[i]$$

**Definition 17** (Key-value store order)**.** A key-value store order $\mathcal{K}_1 \sqsubseteq_{kvs} \mathcal{K}_2$ is defined for two key-value stores $\mathcal{K}_1$ and $\mathcal{K}_2$, and it holds if and only if all the following conditions hold:

$$\forall key.\ \mathcal{K}_1\ key \sqsubseteq_{vl} \mathcal{K}_2\ key$$
$$\textbf{view\_atomic}\ \mathcal{K}_2\ (\lambda key.\ \{0, \ldots, \boldsymbol{length}\ (\mathcal{K}_1\ key) - 1\})$$

In contrast to the other two definitions, the key-value store order requires not only that one key-value store is an extension of another, but also that the expansion is done by applying a finite number of atomic transactions to the key-value store. For the latter to hold, it is enough that $\mathcal{K}_1$ is equivalent to an atomic view of $\mathcal{K}_2$, as stated in the second condition of the key-value store order definition. In other words, the part of $\mathcal{K}_2$ which does not exist in $\mathcal{K}_1$ should be written by fresh transactions that are not already in $\mathcal{K}_1$; otherwise, the atomic visibility is violated because it means the expansion adds the missing part of an incomplete (in progress) transaction.

These ordering relations are especially beneficial for protocol verification using refinement proofs, in particular, for proving that the well-formedness of views is preserved by a transaction commit, which essentially expands the KVS. We also prove that these relations are reflexive and transitive.

### KVS well-formedness proof

We have proved that the four conditions of the KVS well-formedness (Section 3.2) are invariants of the execution test model. Starting from an initial configuration $(\mathcal{K}_0, \mathcal{U}_0) = (\lambda k.\ [v_0], \lambda cl.\ (\lambda k.\ \{0\}))$ which satisfies all conditions and is therefore well-formed, we show the execution test transition (Definition 14), i.e., the transaction commit, preserves the well-formedness. As a result, the KVS of all reachable states are proven to be well-formed.

**snapshot property:** This property is broken into two parts concerning transaction reader-sets and writers. In both parts, we show that a transaction $t$ can not appear in two different versions $i$ and $j$ of the same key as readers or writers, respectively. This is done by case analysis on $i$ and $j$ (on whether they have been added by the commit) and showing a contradiction in all cases where $i \neq j$ because of the TID freshness for the writer of all newly committed versions.

**ww_so property:** We consider two arbitrary versions $i$ and $j$ ($i < j$ without loss of generality) of the same key in the KVS after the commit, whose writers have a WW dependency. We show by case analysis on $i$ and $j$ that in no circumstances can $j$'s writer appear before $i$'s writer in the session order SO, or its reflexive closure (the writers can not be equal as already shown in the snapshot property).

**wr_so property**: We show that for an arbitrary version $i$ in the KVS, and an arbitrary transaction $t$ in its reader-set, $t$ can never appear before version $i$'s writer in the session order or be equal to it. We prove this by case analysis on $i$, showing that the commit can not add such a version to the KVS that violates this property.

**kvs_initialized property:** For this property, we show that the commit does not change the initial version and the length of each version list always either stays the same or increases by one after a commit. So as long as the starting state is initialized, the version lists will remain initialized and non-empty.

## 3.4 Programming language

### Syntax

The core programming language consists of values $v$ and program variables $x$ which build expressions of the form $E ::= v \mid x \mid E + E \mid \cdots$. The syntax of this language is as follows:

$$C ::= skip \mid C_p \mid [T] \mid C;\ C \mid C + C \mid C^* \qquad C_p ::= x := E \mid assume(E)$$

$$T ::= skip \mid T_p \mid T;\ T \mid T + T \mid T^* \qquad T_p ::= C_p \mid x := [E] \mid [E_1] := [E_2]$$

Primitive transactions ($T_p$) can be a lookup ($x := [E]$) or mutation ($[E_1] := [E_2]$). In our modeling, keys are fixed numbers and are not calculated from an expression, so mutations (writes) are modeled as $key := [E]$. Primitive commands ($C_p$) are either an assignment or an

assume statement. A command can be a $[T]$, which denotes an atomic transaction, a sequential composition (;), a non-deterministic choice (+), or an iteration (*) of other commands. Transactions are built similarly using primitive transactions and compound constructs. *skip* is used both as a command and as a transaction.

## Transaction and Command semantics

In this subsection, we explain transaction and command semantics. For modeling client programs, in addition to the abstract model state's (configuration) components, we need client-local stacks for storing values in variables and expression evaluations. The evaluation $[\![E]\!]_s$ of expression E is parametric in the client-local stack s. For a variable $x$, it evaluates to $s(x)$, i.e., the value stored for variable $x$, for values, it evaluates to the same value, and for compound expressions, it distributes on parts of the expression.

**Definition 18** (Transactional step, local state transition)**.** Given a client-local stack $s$ and a (transactional) snapshot $\sigma$, a primitive transaction changes them as follows:

$$(s,\sigma) \xrightarrow{x:=E} (s\ [x \mapsto [\![E]\!]_s],\ \sigma) \qquad (s,\sigma) \xrightarrow{assume(E)} (s,\ \sigma) \text{ if } [\![E]\!]_s \neq 0$$

$$(s,\sigma) \xrightarrow{x:=[E]} (s\ [x \mapsto \sigma\ ([\![E]\!]_s)],\ \sigma) \qquad (s,\sigma) \xrightarrow{[E_1]:=E_2} (s,\ \sigma[[\![E_1]\!]_s \mapsto [\![E_2]\!]_s])$$

**Remark:** In Isabelle/HOL, we have modeled the transactional step with some changes:

$$(s,\sigma) \xrightarrow{Assign\ f} (f\ s,\ \sigma) \qquad (s,\sigma) \xrightarrow{Assume\ t} (s,\ \sigma) \text{ if } t\ s$$

$$(s,\sigma) \xrightarrow{Lookup\ k\ f\_rd} (f\_rd\ (\sigma\ k)\ s,\ \sigma) \qquad (s,\sigma) \xrightarrow{Mutate\ k\ f\_wr} (s,\ \sigma[k \mapsto f\_wr\ s])$$

The *lookup* (x := [E]) primitive transaction is modeled as a constructor with a key $k$ and a function $f\_rd : value \Rightarrow stack \Rightarrow stack$ that maps a value and a stack to a new stack. The new stack after a lookup can then be constructed using this function and the looked-up value $(\sigma\ k)$. Similarly, *mutation* $([E_1] := E_2)$ is modeled using a constructor with a key $k$ and a function $f\_wr : stack \Rightarrow value$ that maps a stack to a value. The new snapshot after a mutation is constructed using this function and the client-local stack $s$ to replace the value of $k$ by the new value $f\_wr\ s$ in the old snapshot $\sigma$.

For primitive commands, we have defined an *assign* constructor with a function $f : stack \Rightarrow stack$ that abstracts the stack update of the assign command. Finally, for *assume* statements, the constructor has an argument $t : stack \Rightarrow boolean$ that, given the stack, evaluates E (checks the assume condition) and returns true or false.

For non-primitive transactions and commands, the complete list of all transition rules, such as standard compound constructs' processing steps, have been discussed in detail in Xiong's PhD Thesis [15]. We have modeled these transition rules as command/transaction steps in

the programming language model in Isabelle/HOL. However, we focus here on explaining the central rule of the framework for committing atomic transactions, which works based on the execution tests that give rise to different consistency models. The equivalence of these execution tests to well-known declarative definitions of consistency models based on abstract executions and dependency graphs has been proven in [15].

**Definition 19** (CAtomicTrans transition rule)**.**

$$\frac{u \sqsubseteq u'' \quad \sigma = \mathit{snapshot}(\mathcal{K},\ u'') \quad (s, \sigma, \emptyset),\ T \rightsquigarrow^*\ (s',\ \_,\ \mathcal{F}),\ \mathit{skip} \quad \mathbf{canCommit_{ET}}\ (\mathcal{K},\ u'',\ \mathcal{F}) \quad t \in \text{NextTxID}\ (cl,\ \mathcal{K}) \quad \mathcal{K}' = \text{UpdateKV}\ (\mathcal{K},\ u'',\ \mathcal{F},\ t) \quad \mathbf{vShift_{ET}}\ (\mathcal{K},\ u'',\ \mathcal{K}',\ u')}{cl \vdash (\mathcal{K},\ u,\ s),\ [T] \xrightarrow{(cl,\ u'',\ \mathcal{F})}_{ET} (\mathcal{K}',\ u',\ s'),\ \mathit{skip}}$$

This rule represents a labeled transition over a state, comprising a key-value store $\mathcal{K}$, a client view $u$, and a client-local stack $s$. It shows the transition premises for a client $cl$ committing an atomic transaction $T$. Well-formedness of the mentioned views is implicitly assumed.

The transition is parametric in $ET$, the execution test, which can generate various consistency models for different instantiations. The transition label consists of the client identifier $cl$, an intermediate view $u''$, and the transaction fingerprint $\mathcal{F}$.

The client's view, in the beginning, is $u$. The transition consists of two parts; first, the client's view $u$ advances to a view $u''$, capturing the fact that a client receives synchronization messages before committing a transaction; second, the transaction is committed in $\mathcal{K}$ with a final fingerprint $\mathcal{F}$ under the view $u''$, updating the key-value store to $\mathcal{K}'$ and shifting the view to $u'$. The premises on the snapshot, $\mathbf{canCommit_{ET}}$, and $\mathbf{vShift_{ET}}$ are concerned with the second part of the transition; therefore, they use $u''$.

The fingerprint $\mathcal{F}$ is generated by doing a pre-calculation (local execution) on transaction $T$. A multi-step transition, denoted by $\rightsquigarrow^*$, over a client stack, a snapshot, and a fingerprint, which starts with an empty fingerprint and the transaction $T$ and executes the transaction step by step, updating the fingerprint with the read or write on each step, and keeping the remaining parts of the transaction to be executed after each step, until it reaches skip, which means the transaction execution has been completed and the generated fingerprint is final. The single transactional step is defined in Definition 18.

The transaction identifier $t$ is chosen from a set of fresh transaction IDs (NextTxID) and used in UpdateKV (defined below) to update the KVS with the fingerprint $\mathcal{F}$.

**Definition 20** (Transactional update)**.** UpdateKV $(\mathcal{K},\ u'',\ \mathcal{F},\ t)$ is defined as:

$$\text{UpdateKV}\ (\mathcal{K},\ u'',\ \emptyset,\ t) \triangleq \mathcal{K}$$

$$\text{UpdateKV}\ (\mathcal{K},\ u'',\ \{(\mathrm{R}, k, v)\} \uplus \mathcal{F},\ t) \triangleq \mathit{let}\ i = \mathit{max}_<(u''(k))\ \mathit{and}\ (v, t', T) = \mathcal{K}(k, i)\ \mathit{in}$$
$$\text{UpdateKV}\ (\mathcal{K}[k \mapsto \mathcal{K}(k)[i \mapsto (v, t', T \uplus \{t\})]],\ u'',\ \mathcal{F},\ t)$$

$$\text{UpdateKV}\ (\mathcal{K},\ u'',\ \{(\mathrm{W}, k, v)\} \uplus \mathcal{F},\ t) \triangleq \mathit{let}\ \mathcal{K}' = \mathcal{K}[k \mapsto \mathcal{K}(k) :: (v, t, \emptyset)]\ \mathit{in}$$
$$\text{UpdateKV}\ (\mathcal{K}',\ u'',\ \mathcal{F},\ t)$$

where for more readability, a version $(\!|\ v\_value = v,\ v\_writer = t,\ v\_readerset = T\ |\!)$ is written as $(v, t, T)$.

The UpdateKV function, for each of the fingerprint's read operations, adds the transaction identifier $t$ to the reader-set of the read version, which must be the last version in the view $u''$. For each of the fingerprint's write operations, the UpdateKV function appends a new version ($v$, $t$, $\emptyset$) to the version list of the key $k$ in $\mathcal{K}$, where $k$ and $v$ are the recorded key-value pair in the fingerprint, $t$ is the writer transaction, and $\emptyset$ indicates the empty reader-set of the newly written version.

## Program semantics

**Definition 21** (PProg transition rule)**.**

$$\frac{u = \mathcal{U}(cl) \qquad s = \mathcal{E}(cl) \qquad c = \mathrm{P}(cl) \qquad cl \vdash (\mathcal{K},\ u,\ s),\ \mathrm{C} \xrightarrow{\lambda}_{ET} (\mathcal{K}',\ u', s'),\ \mathrm{C}'}{\vdash\ (\mathcal{K},\ \mathcal{U},\ \mathcal{E},\ \mathrm{P}) \xrightarrow{\lambda}_{ET} (\mathcal{K}',\ \mathcal{U}[cl \mapsto u'],\ \mathcal{E}[cl \mapsto s'],\ \mathrm{P}[cl \mapsto \mathrm{C}'])}$$

This rule lifts the execution of a client's program step to the level of the whole system, whose state $(\mathcal{K},\ \mathcal{U},\ \mathcal{E},\ \mathrm{P})$ consists of a configuration $(\mathcal{K},\ \mathcal{U})$, a client environment $\mathcal{E}$, which is a function from client identifiers to client-local stacks, and a program $\mathcal{P}$, which maps client identifiers to commands (each client program formulated as one command). This rule captures the execution of program P for one step, i.e., one step for one of the clients' commands which results in program $\mathrm{P}[cl \mapsto \mathrm{C}']$, where $\mathrm{C}'$ is the result of a command step for client $cl$. The label $\lambda$ is either of the form $(cl,\ u",\ \mathcal{F})$ for an atomic transaction (CAtomicTrans rule), or a single client identifier $cl$ otherwise, in case the command does not access $\mathcal{K}$.

## KVS well-formedness proof

Finally, we prove the well-formedness of the KVS for the programming language model by showing that the programming language model refines the abstract execution test model. This implies that properties, including KVS well-formedness, are preserved from the execution test model to the programming language model.

The challenging cases for the proof were the *atomic transaction* and *sequential composition*. For the atomic transaction, we proved a lemma showing that the snapshot premise ($\sigma = snapshot(\mathcal{K},\ u")$) and the multi-step local transaction execution for fingerprint generation ($((s, \sigma, \emptyset),\ T \rightsquigarrow^*\ (s',\ \_,\ \mathcal{F}),\ skip$) imply the $fingerprint\_property$. For sequential composition, we proved two lemmas for the two cases of the $\lambda$ labels and what they imply about the command type. The Isabelle proof is available in the thesis's GitLab repository, linked in Appendix.

## 3.5 Protocol Verification by Refinement

For protocol verification in our framework, we use refinement. Since refinement guarantees property preservation, if we prove that a protocol refines a consistency model (abstract model obtained by instantiating the execution test), then the properties of the consistency model also hold for the protocol (the concrete model). As a result, we can prove that a protocol satisfies a consistency model by refinement.

The protocol verification by refinement consists of two steps:

- modeling the protocol as an event system.
- proving that the modeled protocol refines the execution test of the desired consistency model.

In order to prove the refinement, the following conditions must hold:

- For the events that refine skip, the reconstructed abstract state, i.e., the client views and the global KVS do not change.
- For the events that refine the commit event of the abstract model, the following properties must hold:
    1. Client views stay wellformed through these events.
    2. $canCommit_{ET}$ of the desired consistency model holds for the reconstructed KVS and client views before and after the event.
    3. $vShift_{ET}$ of the desired consistency model holds for the client view update.
    4. The views always only grow.
    5. The transaction ID is fresh.
    6. The reconstructed key-value store after the event is the updated version of the reconstructed key-value store before the event.
    7. The only client view that has changed is the view of the client who owns the transaction,
    8. For the reads in fingerprints, the read version corresponds to the last version available in the client view.

# Chapter 4

# Verification of Two-Phase Locking Protocol - Serializability

The first protocol that we verified is the Two-Phase Locking (2PL) protocol, which is combined with a Two-Phase Commit (2PC) protocol. This is a very common approach for databases that need strong consistency guarantees.

A 2PC protocol works on a distributed system with a Transaction Manager (TM) and several Resource Manager (RM) machines. In our model, each RM only handles the reads and writes to one key, hence the name "Key Manager (KM)" is given to it.

In a 2PL protocol, each transaction acquires all of its necessary locks before changing anything in the database. For each key, depending on whether the transaction reads and/or writes to that key, the type of the lock the corresponding KM has to acquire is determined: a read lock for a read, a write lock for a write or a read-and-write, and no lock in case the key is not involved in the transaction. For a given key $k$, write locks are exclusive, while read locks can be obtained by multiple transactions. So for example, if a transaction $t$ requires a write lock for a key $k$ but a read lock or write lock on $k$ has already been acquired by another transaction $t'$, $t$ will abort. However, if a transaction needs a read lock for a key, it can get it even if other transactions have a read lock on that key. It only needs to abort if a transaction has a write lock on the key, which means no new locks can be put on that key. If all of the KMs succeed in obtaining the necessary locks for a transaction, that transaction will be executed and committed. Otherwise, even if one of the glsKMs goes to the "Not Okay" state (meaning the lock was not available or there was an internal crash), the whole transaction will be aborted, and all the already acquired locks will be released.

In our proof, using the mechanized framework, we show that this protocol satisfies the SER consistency model, so there exists a total order on all transactions. The SER consistency model is defined by instantiating the execution test with $R_{SER} = (\bigcup_k WW_{\mathcal{K}}(k))^{-1}$ as $R_{ET}$ where $\mathcal{K}$ is the KVS, and True as vShift (See Section 3.1 for definitions).

We will not discuss the correctness of such instantiations, as this has already been addressed in Xiong's PhD thesis [15]. Nevertheless, to get an intuition about why the two-phase locking

protocol satisfies the serializability consistency model we should take a look at the definition of WW dependency relation and what $R_{ET} = (\bigcup_k WW_K\ (k))^{-1}$ implies. The *canCommit* condition in the execution test requires that the visible transactions are closed under $R_{ET}$. This means if a transaction $t$ is visible for a given key $k$, then another transaction $t'$ that has a WW relation with it $(t', t) \in WW_K\ (k)$ should also be visible, and this, in turn, implies that every $t''$ such that $(t'', t') \in WW_K\ (k)$ should also be visible and so on. So for each key, the view should be a complete (full) view, containing all versions from the initial version up to the version that the client is currently working on.

The 2PL protocol satisfies this condition because only one transaction can add versions to the KVS at a time. So by growing the client views to include the full view on the KVS each time before they commit, we ensure that the client views will always stay complete, up to the version they are committing.

To see in more detail how the actual proof works, we will first define the global state and the events of the event system, and then present the relation function and mediator function used for the refinement and the reasoning for the interesting cases.

## 4.1 States

In our model, each client corresponds to a TM, and each KM handles the reads and writes to one key. The global state contains a mapping from *cl_id*s (client identifiers) to transaction managers' states and a function from keys to key managers' states:

- TM state consists of a status which can be *Init, Prepared, Committed, or Aborted*, called *tm_status*, a variable *tm_sn* which keeps the last sequence number used by the client, and a variable *tm_view* that keeps track of the client view. The *tm_sn* together with the *cl_id*, determines the transaction TID being processed by the TM. In the beginning, *tm_sn* is set to 0 and after each round, which is marked by returning to the Init status, *tm_sn* is increased by one.

- KM state contains a version list for the corresponding key, called *km_vl*. For each transaction id, the KM has a different status and a fingerprint, so a function from TIDs to KM statuses are kept in *km_status* and similarly a function from TIDs to key fingerprints in *km_key_fp*. The KM status can have any of the values *Working, Prepared, ReadLock, WriteLock, NoLock, NotOkay, Committed, or Aborted*. A key fingerprint is represented by a mapping from R/W to a value option type.

Figure 4.1 shows the state diagrams of a TM state and a KM state. Keep in mind that there exist one such TM state machine for each client *cl*, and one such KM state machine for each key and TID pair ($k$, $t$).

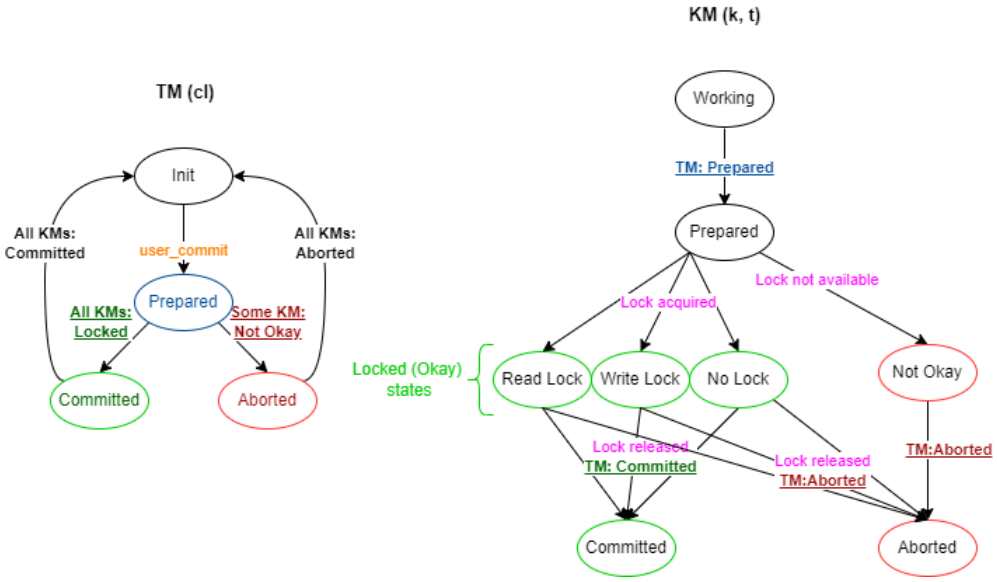


Figure 4.1: Two-Phase Locking + Two-Phase Commit State Diagrams

## 4.2 Events

We first give the formal definitions of the events and describe them further below.

**Transaction Manager Events**

- **user_commit(*s, s', cl*):**
  $tm\_status(cl) = TM\_Init \triangleright tm\_status(cl) := TM\_Prepared$

- **tm_commit(*s, s', cl, sn, u'', F*):**
  $tm\_status(cl) = TM\_Prepared \wedge sn = tm\_sn(cl) \wedge$
  $u'' = (\lambda k.\ full\_view\ (\boldsymbol{kvs\_of\_gs}\ s\ k)) \wedge F = (\lambda k.\ km\_key\_fp\ (k,\ get\_txn\_cl\ cl)) \wedge$
  $(\forall k.\ km\_status(k,\ get\_txn\_cl\ cl) \in \{ReadLock,\ WriteLock, NoLock\}) \triangleright$
  $tm\_status(cl) := TM\_Committed \wedge tm\_view(cl) := (\lambda k.\ full\_view(\boldsymbol{kvs\_of\_gs}\ s'\ k))$

- **tm_abort(*s, s', cl*):**
  $tm\_status(cl) = TM\_Prepared \wedge (\exists k.\ km\_status(k, get\_txn\_cl\ cl) = NotOkay) \wedge$
  $(\forall k.\ km\_status(k, get\_txn\_cl\ cl) \in \{ReadLock,\ WriteLock, NoLock, NotOkay\}) \triangleright$
  $tm\_status(cl) := TM\_Aborted$

- **tm_ready_c(*s, s', cl*):**
  $tm\_status(cl) = TM\_Committed \wedge (\forall k.\ km\_status(k, get\_txn\_cl\ cl) = Committed) \triangleright$
  $tm\_status(cl) := TM\_Init \wedge tm\_sn(cl) := \boldsymbol{Suc}\ (tm\_sn(cl))$

- **tm_ready_a(*s, s', cl*):**
  $tm\_status(cl) = TM\_Aborted \wedge (\forall k.\ km\_status(k, get\_txn\_cl\ cl) = Aborted) \triangleright$
  $tm\_status(cl) := TM\_Init \wedge tm\_sn(cl) := \boldsymbol{Suc}\ (tm\_sn(cl))$

**Key Manager Events**

- **prepare(*s, k, s', t*):**
  $tm\_status(get\_cl\_txn\ t) = TM\_Prepared\ \wedge\ km\_status(k,\ t) = Working\ \triangleright$
  $km\_status(k,\ t) := Prepared$

- **acquire_rd_lock(*s, k, v, s', t*):**
  $tm\_status(get\_cl\_txn\ t) = TM\_Prepared\ \wedge\ (\forall t'.\ km\_status(k,\ t') \neq\ WriteLock)\ \wedge$
  $km\_status(k,\ t) = Prepared \wedge v = v\_value\ (\textbf{\textit{last\_version}}\ (km\_vl(k))\ (\textbf{\textit{full\_view}}\ (km\_vl(k))))\ \triangleright$
  $km\_key\_fp(k,\ t)\ W := \textbf{\textit{None}}\ \wedge\ km\_key\_fp(k,\ t)\ R := \textbf{\textit{Some}}\ v\ \wedge$
  $km\_status(k,\ t) :=\ ReadLock$

- **acquire_wr_lock(*s, k, v, ov, s', t*):**
  $tm\_status(get\_cl\_txn\ t) = TM\_Prepared \wedge (\forall t'.\ km\_status(k,\ t') \notin \{WriteLock,\ ReadLock\}) \wedge$
  $km\_status(k,\ t) = Prepared\ \wedge$
  $(ov = \textbf{\textit{None}} \vee ov = \textbf{\textit{Some}}\ (v\_value\ (\textbf{\textit{last\_version}}\ (km\_vl(k))\ (\textbf{\textit{full\_view}}\ (km\_vl(k))))))\ \triangleright$
  $km\_key\_fp(k,\ t)\ W := \textbf{\textit{Some}}\ v\ \wedge\ km\_key\_fp(k,\ t)\ R := ov\ \wedge$
  $km\_status(k,\ t) :=\ WriteLock$

- **acquire_no_lock(*s, k, s', t*):**
  $tm\_status(get\_cl\_txn\ t) = TM\_Prepared\ \wedge\ km\_status(k,\ t) = Prepared\ \triangleright$
  $km\_key\_fp(k,\ t)\ W := \textbf{\textit{None}}\ \wedge\ km\_key\_fp(k,\ t)\ R := \textbf{\textit{None}}\ \wedge$
  $km\_status(k,\ t) := NoLock$

- **nok (*s, k, s', t*):**
  $tm\_status(get\_cl\_txn\ t) = TM\_Prepared\ \wedge$
  $km\_status(k,\ t) = Prepared\ \triangleright\ km\_status(k,\ t) := NotOkay$

- **commit(*s, k, s', t*):**
  $km\_status(k,\ t) \in \{ReadLock,\ WriteLock, NoLock\}\ \wedge$
  $tm\_status(get\_cl\_txn\ t) = TM\_Committed\ \triangleright$
  $km\_vl(k) := \textbf{\textit{update\_kv\_key}}\ t\ (km\_key\_fp\ (k,\ t))\ (\textbf{\textit{full\_view}}\ (km\_vl(k)))\ (km\_vl(k))\ \wedge$

  $km\_status(k,\ t) := Committed$

- **abort(*s, k, s', t*):**
  $km\_status(k,\ t) \in \{ReadLock,\ WriteLock, NoLock, NotOkay\}\ \wedge$
  $tm\_status(get\_cl\_txn\ t) = TM\_Aborted\ \triangleright\ km\_status(k,\ t) := Aborted$

A summary of all events is given above. The ***last_version*(*vl, u*)** function, used by some events, returns the most recent version of the version list $vl$ whose version number is included in the view $u$. If $u$ is a ***full_view*** of $vl$, then ***last_version*(*vl, u*)** will return the last version of

the entire version list $vl$. The function $\boldsymbol{update\_kv\_key(t, Fk, uk, vl)}$ commits a read and/or a write to a key's version list $vl$ (See Definition 20). The argument $t$ represents the fresh TID of the read/write transaction, $Fk$ represents the key fingerprint, which stores the read and write that will be committed, and $uk$ is the view of the client on the key, whose maximum version number determines the version read from $vl$.

The sequence of TM and KM events for executing a transaction $t$ is as follows:

With the *user_commit* event, the TM of the client who owns t (captured with the *get_cl_txn* function), signals all key managers to prepare for a commit. All KMs follow with the *prepare* event, moving from the *Working* to the *Prepared* state for transaction $t$. Keep in mind that the *km_status* for other transactions on the same KM will not change. The prepared KMs can then proceed to attempt acquiring the locks they need depending on the operation (read/write) that they are going to perform, or no lock at all if they are not included in the transaction. This information is stored in the key fingerprint variable on each KM and is assumed to be known for each individual key and transaction by a prior analysis of the transaction. This sudden appearance of data may be later refined to an event that transfers such information from the TM to all corresponding KMs, but for now, we keep it simple by assuming this data is available. Each KM might change its *km_status* of transaction t from the *Prepared* state to any of the *ReadLock*, *WriteLock*, and *NoLock* states if it succeeds in acquiring the necessary locks, and to the *NotOkay* state if the necessary locks are not available or if it crashes internally.

The TM will then decide to commit or abort depending on the status of all KMs. If at least one KM is in the *NotOkay* state, the TM will abort (*tm_abort* event), otherwise, *tm_commit* is executed, which is the event that simulates the commit of the abstract model, i.e., execution test. At this point, we know for sure whether all KMs will abort or commit based on the TM state. This is why we can already reconstruct the abstract state, consisting of the global KVS and client views, before the KMs actually commit (See Section 4.3). After all KMs have followed the commit or abort, the TM goes back to the *Init* state and increments the sequence number which is stored for the corresponding client (events *tm_ready_c* or *tm_ready_a*).

It should be noted that the view updates and assignments in *tm_commit* are intended to directly simulate the changes made to a client's view during an abstract commit event. It is therefore necessary for the client view to first grow into an intermediate $u''$ view, which must encompass all the versions that are currently assumed to be committed. To accomplish this, the corresponding abstract global KVS of the current state $s$ can be reconstructed using $\boldsymbol{kvs\_of\_gs}$ on s. A more detailed description of this function will be provided in the next section. Finally, having a full view of the current global KVS, the commit is executed, and the view is updated to include the changes made by the transaction as well. Thus we use $\boldsymbol{kvs\_of\_gs}$ again, this time called on the resulting state, $s'$, which contains the newly committed versions. This does not cause a circular update, because $\boldsymbol{kvs\_of\_gs}$ does not depend on the view.

## 4.3 Refinement

For the verification of the protocol, we use refinement in Isabelle/HOL. In order to show that the protocol satisfies a consistency model, we need to show that it refines the execution test of that consistency model.

For the refinement from the abstract model (execution test) to the concrete model (protocol), we define a mapping from the concrete state to the abstract state, called a simulation function, which reconstructs the corresponding abstract state at each point in the protocol. Furthermore, we introduce a mediator function, which is a mapping from concrete events to abstract events.

### Simulation and mediator functions

**Simulation Function:** The abstract state is a *config*, containing a global KVS and client views. So the simulation function is defined with the help of the two following functions that reconstruct these two parts:

- ***kvs_of_gs*:** This function builds the global KVS of the abstract state by combining the version lists of all keys that are stored on KMs. It updates these version lists' content by using the functions that are shown in Table 4.1.

| Functions | Effect in the abstract state reconstruction process |
|---|---|
| ***update_kv_reads_all_txn*(*vl*)** | Updates the **readerset** of the **last version** in *vl* with **pending reads** of all prepared-to-commit transactions. |
| ***update_kv_writes_all_txn*(*vl*)** | Appends to *vl*, the **pending write** of a single writer-transaction that is prepared to commit (Only one such transaction exists because of the **write lock** on the key). |
| ***update_kv_all_txn*(*vl*)** | Applies the two functions above on a given *vl* of a key. |

Table 4.1: Functions used in the refinement's Simulation function

  *kvs_of_gs* effectively calls the *update_kv_all_txn* on all key's version lists while passing the TM and KMs states' information and the keys' fingerprints to the function as well.

- ***views_of_gs*:** This function builds the client views of the abstract state. It simply combines the *tm_view* component of all clients' TM state. The *tm_view* component keeps track of the client's view during the protocol and is only updated in the *tm_commit* event where it is set to include all versions of the reconstructed abstract state after the commit. This ensures that at each point of the protocol the *tm_view* represents the expected view of the client from the perspective of the abstract model.

**Mediator Function:** As previously mentioned, the *tm_commit* event simulates the abstract commit in this protocol. All other events refine *Skip*, where the state stays the same.

### The refinement proof

We take care of the fact that all effects of a transaction (in KVS and client views) are only visible once the TM has entered the *TM_Committed* state. None of the other events should cause a change in the reconstructed abstract state, so we must make sure, once the transaction effects are reflected in the reconstructed abstract global KVS and client views, individual commits of the KMs will not change that. In other words, by taking note of the *km_status* variable, the pending commits are prematurely applied to the KVS and client views, and the modifications of already committed KM are not applied again. This results in the same abstract state after the TM commits, regardless of how many of the KMs' commits have followed.

The refinement proof uses numerous invariants and lemmas that describe the behavior of the system. These include but are not limited to the following categories:

- **Lock invariants:** These invariants establish the semantics of read locks and write locks. *RLockInv* states that if a transaction $t$ has a read lock on a key $k$, i.e., the corresponding KM status for this transaction is $km\_status(k, t) = ReadLock$, no other transactions can have a write lock on that key. *WLockInv* states that at most one transaction possesses the write lock for a key $k$. The KM status of such a transaction $t$ is $km\_status(k, t) = WriteLock$.

- **System state for past and future TIDs:** These invariants are needed to show the status of KMs for the transactions that were running in the past (have a smaller sequence number than the current transaction, stored in *tm_sn*), or will be executed in future (larger sequence number than *tm_sn*). For past transactions, the status is always either *Committed* or *Aborted* depending on the transaction's success in committing. For future transactions, the status of all the KMs is *Working*, because they are ready to start.

- **TID freshness:** For this section, we have one invariant and several lemmas to show that the current transaction has a fresh TID and doesn't appear in the KVS before the commit. The invariant *SqnInv* states that before committing ($tm\_status(cl) \neq TM\_Committed$), all existing sequence numbers for client *cl* in the KVS are strictly smaller than the value of *tm_sn*, and after committing ($tm\_status(cl) = TM\_Committed$), all existing sequence numbers are smaller or equal to *tm_sn*, since now the transaction with this sequence number can be present in the KVS if the transaction has succeeded to commit. After the transaction is finished, *tm_sn* is incremented by 1, so the invariant's condition is preserved. The lemmas cover the different scenarios that arise from the type of transactions, the commits and aborts of the transactions, and their effect on the sequence numbers of the relevant client and other non-affected clients that work on KVS.

- **KVS invariants:** The key-value store has some properties that are addressed by these invariants. *KVSNonEmp* shows that the version lists of all KMs are always non-empty, and

this is true because they are all initialized with a $v_0$ version and only grow. *KVSGSNonEmp* shows that the reconstructed KVS in the simulation function is also always non-empty as it is built based on the version lists and contains at least the $v_0$ committed versions. *KVSLen* states that the length of the reconstructed KVS for a certain key is always larger than or equal to the length of the version list of that key, which is stored in $km\_vl$. This holds because the reconstructed state might also include the pending writes that do not appear in the version list yet but are going to be committed since the TM has committed. There are also some lemmas about how different events preserve or change the reconstructed KVS.

- **Fingerprint content invariants:** There are three invariants in this section, for each of the locking conditions: read lock, write lock or no lock at all. These invariants establish the conditions on the key fingerprint's content for each of these cases. A read lock means the fingerprint contains a read on the key and no write. A write lock, however, only requires the fingerprint to have a write but does not put a condition on the read part of the fingerprint. The no-lock state means the fingerprint has to be empty. Furthermore, two other invariants are specifically defined about the content of the fingerprint's read part, ensuring that it follows the rule that the fingerprint always contains the latest version of the view.

- **View updates and view well-formedness:** Invariant *KVSView* is concerned with the view well-formedness of $tm\_view$ variable in all states with respect to the reconstructed KVS ($kvs\_of\_gs$). To prove this, we show that each event only causes the reconstructed KVS to grow while preserving the view well-formedness, i.e., key-value store order (Look at Definition 17) $kvs\_of\_gs\ s \sqsubseteq_{kvs} kvs\_of\_gs\ s'$ holds. Invariant *TMFullView*, which establishes the order between the $tm\_view$ variable and a full view on $kvs\_of\_gs$ is proven by showing the view growth through an event for a full view on $kvs\_of\_gs$ and a full view on $km\_vl$.

A detailed list of all invariants and important lemmas of the refinement proof is given in Appendix A.

# Chapter 5

# Verification of the Eiger-PORT+ Protocol - Causal+ Consistency

Our second candidate for verification is the Eiger-PORT protocol, a performance-optimal protocol for read-only and write-only transactions which uses version clocks and satisfies (transactional) CC consistency model [2].

## 5.1 The Eiger-PORT Protocol

The Eiger-PORT protocol is an optimized version of an older protocol that supports read-only and write-only transactions, called Eiger [22]. Eiger uses Lamport clocks [17] to assign a timestamp to each committed write. *Write transactions* in Eiger follow a special variation of the 2PC protocol that always commits (no abort state for either TM or RM). Read-only transactions in Eiger are non-blocking, might need one to three rounds of communication before committing, and have linear metadata with respect to the number of conflicting write transactions. Eiger-PORT uses the same mechanism as Eiger for the write transactions but optimizes the read-only transactions by using only one round of communication and constant metadata.

In Eiger-PORT, each server has a local safe time (lst) that corresponds to the minimum of pending transactions' timestamps, or if there is none, the maximum committed timestamp used on that server. Each client keeps track of these $lst$s in a $lst\_map$ variable that is a mapping from server identifiers to their $lst$, and a global safe time (gst) which is always the minimum timestamp of $lst\_map$ and works as the *stable frontier* (the maximum timestamp that is guaranteed to point to a committed transaction for all servers) for that client.

The protocol updates the value of the $gst$ variable before each *read-only transaction* to reflect the minimum value in the $lst\_map$. For each read, a read request at the read timestamp equal to the current $gst$ is sent to the responsible server. The server first reads at the given time stamp and retrieves the last available version committed before or at the read timestamp. Then it proceeds to do a full scan on all versions that are committed *after* the retrieved version on the server and checks if any of these are written by the same client. If such a version is found,

it is returned to the client to satisfy the Read Your Writes (RYW) consistency guarantee. If no such version is found, the server first checks if the initially retrieved version is written by the same client. If not, it will just return that version; otherwise, a *find_isolation* function is called that looks for any versions that have committed *before* this version but *after* the version's *gst* (The snapshot that the transaction's writer client depended on), and if it finds a match it will return that version instead. This can potentially result in different orderings of versions from the perspective of different clients (client views), called *per-client ordering*.

## 5.2 Protocol Formalization Overview

During formalization of the operational framework of Xiong et al. [10], we noticed that it implicitly assumes a total order on non-causally-ordered writes to the same key, i.e., all clients see the versions of a key's version list in the same order since all client views refer to indices of the version list to uniquely identify versions. This makes all the consistency models defined on top of this framework operate on the same assumption. This is especially important for weaker consistency models such as CC that do not have this assumption in their definition, and the versions can be ordered differently for different sessions. For the CC case, however, there is a stronger consistency model in the literature, namely the Causal+ Consistency (CC+) model, also called Causal Convergence (CCv), that requires precisely this assumption in addition to the plain CC requirements. So, in fact, the operational framework of Xiong et al. [10] models CC+, instead of the plain CC. This distinction is vital since CC+ is the de facto of production database systems and has been adapted instead of CC in systems such as Cure [23] and MongoDB [24].

The Eiger-PORT protocol [7] satisfies plain CC while its predecessor Eiger satisfies CC+. As we were more interested in verifying the newer and performance-optimized Eiger-PORT protocol, we decided to try further optimizing the Eiger-PORT protocol in a way that satisfies CC+. The modified version of the protocol, which we call Eiger-PORT+, satisfies a stronger consistency model. This modification removes the usage of *find_isolation*, so it also makes the protocol computationally less demanding; therefore, we expect the new protocol to perform better than the original Eiger-PORT protocol. As future work, we plan to show the degree of this performance improvement by implementing Eiger-PORT+ based on the Eiger-PORT code base (available at [25]) and comparing it with Eiger-PORT.

To show that the optimized protocol does not break the write isolation and preserves the atomic visibility, we shall review the definition of atomic visibility based on the anomaly it avoids, namely *fractured reads*. According to the [1], a *fractured read* phenomenon is defined as:

**Definition 22** (Fractured Reads)**.** A transaction $T_j$ exhibits the fractured reads phenomenon if transaction $T_i$ writes versions $a$ of $x$ and $b$ of $y$ (in any order, where $x$ and $y$ may or may not be distinct items), $T_j$ reads version $a$ of $x$ and version $c$ of $y$, and $c < b$.

This means reading two versions from two different keys does not result in a fractured read, as long as the writer transaction of one has not written a newer version of the other key.

In the following paragraphs, we give an example that addresses the interesting case in which the two algorithms differ and elaborate on the changes that we made to the protocol and their effect, using the example.

**Example 1.** There are two servers, $S_A$ responsible for the key $A$, and $S_B$ responsible for the key $B$. At timestamp 0, these keys are initialized with versions A0 and B0. Then at timestamp 3, a write transaction $t_1$ by client $cl_1$ commits A1 and B1 on the servers, and at timestamp 5, a write transaction $t_2$ by client $cl_2$ commits B2 on $S_B$. In Figure 5.1, the state of the keys and their versions on the servers are visualized. On the right is a more detailed representation of the same state, with the hatched part showing the pending period of the write and the solid part showing a committed write, beginning at the commit timestamp and continuing until the next version is committed.

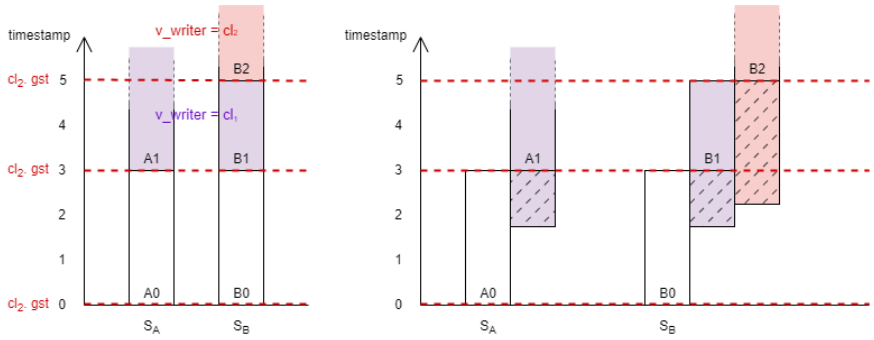


Figure 5.1: Servers and versions of Example 1: Left) The committed versions on the servers; red dashed lines show the read timestamps. Right) Detailed illustration of the versions' start timestamp, pending period (hatched), commit timestamp, and validity period (solid).

Now if $cl_2$ executes a read transaction $R$ that reads from both servers with a *gst* (stable frontier) of 0, which is the minimum timestamp across all servers that the client is aware of, it should read B2 on server $S_B$ to satisfy RYW because it is a version written by $cl_2$. When $R$ tries to read at 0 from $S_B$, it first retrieves B0 and then goes through all the versions to check if there are any newer versions written by $cl_2$, so it finds B2 at timestamp 5 and returns it. This effectively makes $R$ read at timestamp 0 on $S_A$, while reading at timestamp 5 on $S_B$. This is consistent because it effectively "pulls" version B2 back to the stable frontier, e.g., back to timestamp 0. $cl_2$'s read timestamp (gst) will gradually advance and eventually hit B2 again at timestamp 5. This is where the two versions of the protocols differ.

With the original Eiger-PORT protocol, reading at timestamp 5 returns B1 to the client $cl_2$; According to the *find_isolation* function, if B1 has a larger commit time than the *gst* of B2, i.e., the writes of B1 and B2 are conflicting, as shown in Figure 5.2, the protocol returns B1, a version written by another client, rather than returning the client's own write B2 again.

This has been claimed to be necessary to preserve write isolation, i.e., getting an atomic view of transactions.

This decision effectively reorders the versions on B for the client $cl_2$ while $cl_1$ will still see the original ordering. This is allowed by causal consistency since causal consistency does not require a total order.

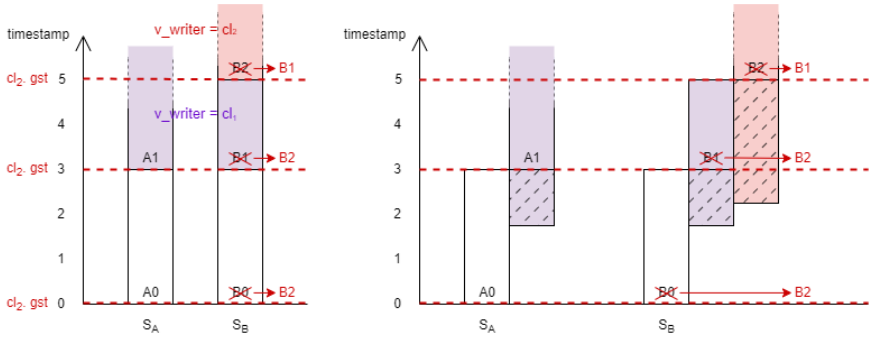


Figure 5.2: Eiger-PORT, client $cl_2$'s perspective of Example 1

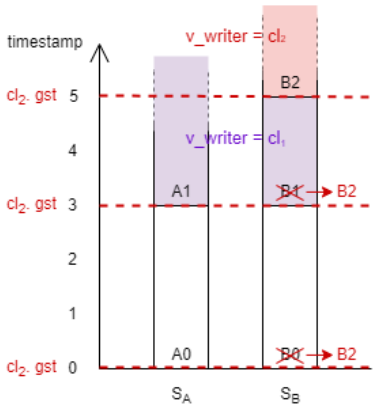


Figure 5.3: Eiger-PORT+, client $cl_2$'s perspective of Example 1

The modification we made on Eiger-PORT to get Eiger-PORT+ is skipping the *find_isolation* step, which means in the example above, $cl_2$ will still read B2 on $S_B$ at timestamp 5. As a result, client $cl_2$ never reads B1, but this is not a problem because as long as the system does not exhibit a fractured read, the *read atomicity* is preserved, even if not all versions of keys are read by one client. So, for example, reading A0 and B2 or A1 and B2 by a client does not violate read atomicity, while reading A1 and B0 or A0 and B1 would be examples of a fractured read. The benefit of this approach is that the computationally demanding function *find_isolation* is omitted, and we regain the total ordering on versions, as the version ordering in client $cl_1$'s view, B0 $\rightarrow$ B1 $\rightarrow$ B2, does not conflict with the version ordering on client $cl_2$'s view, B0 $\rightarrow$ B2. The outcome of a read transaction by $cl_2$ using Eiger-PORT+, at timestamps 0, 3, and 5 for our example, is illustrated in Figure 5.3.

## 5.3 States

The system comprises clients and servers. Each client runs at most one transaction at a time and keeps the state of that transaction in its state. We assume that each server is responsible for a single key and the read and write transactions targeting it. At the model level, having single-key servers is a reasonable assumption and is supported by two reasons. First, we can always combine keys and process them on one server later, while still considering each key and its state as a logical partition. This leads to effectively running the same protocol. Second, even if we want to have a multi-key server with only one copy of the *clock* and *lst*, for this particular protocol, it means that the *lst*, which is the minimum of all pending transactions' timestamps, might stay lower compared to the single-key-server version of the protocol, and as a result, the stable frontier is pulled lower and we might only read staler data. Hence, if we can prove the single-key-server satisfies the CC+ consistency model, the multi-key version will also satisfy it.

Versions in Eiger-PORT+ are modeled with five attributes: *v_value*, *v_writer*, and *v_readerset* are shared with the abstract model's version; *v_is_pending* and *v_ts* are specific to the protocol. *v_ts* can either mean the pending or commit timestamp depending on the value of *v_is_pending*.

The modeling of the system state contains a mapping from *cl_id*s (client identifiers) to client states and from *svr_id*s (server identifiers) to server states.

**Client state:**

- ***txn_state*:** The state of the client's current transaction. It can have four different values: *Idle* when no transaction is running. *RtxnInProg keys kv_map* for read-only transactions where *keys* is the set of keys to be read, and *kv_map* stores the read values for each key as the transaction progresses. *WtxnPrep kv_map* and *WtxnCommit ts kv_map* for write-only transactions, representing the TM prepared and committed states of a 2PC protocol. *kv_map* stores the key-value pairs to be written, and *ts* holds the commit timestamp.
- ***txn_sn*:** A counter, used as a sequence number for this client to generate the TID.
- ***gst*:** Stores the global safe time or *stable frontier* of the client.
- ***lst_map*:** A mapping from *svr_id*s to their *lst*s, as latest known by the client.
- ***cl_view*:** Keeps track of the client view, corresponding to the client view in the abstract model.

**Server state:**

- ***wtxn_state*:** A mapping from TIDs to the state of that transaction for this server. The transaction state on the server can take three different values: *Ready* if the transaction has not started or it is a read transaction. *Prep ts v_id* if the server is in the prepared state for that transaction, and *Commit* if it is in the committed state. These represent resource managers' prepared and committed states in a 2PC protocol. The argument *ts*

is the prepared timestamp of the server for the transaction and *v_id* shows the version identifier of the prepared version.

- ***clock*:** The Lamport Clock [17] of the server.
- ***lst*:** Stores the local safe time of the server.
- ***pending_wtxns*:** A mapping from TIDs to a timestamp option type. If the TID is a pending write transaction, it maps to a *Some ts*, which keeps the transaction's pending timestamp, otherwise, it maps to *None*.
- ***DS*:** Stores the version list of the key, which the server is responsible for.

Figure 5.4 shows the state diagrams of the client's *txn_state* and the server's *wtxn_state*, where the transition labels correspond to the system events. Additionally, on the right, we have shown a virtual state machine for read transactions, similar to the one for write transactions and their commit, only to visualize the effect of the *read registration* event, which adds a transaction to the reader-set of the version it has read. However, this state is not modeled as a variable and is instead handled by looking for the transaction's ID in the versions' reader sets to determine if it has been registered. This prevents having a redundant state variable that makes further refinements of the model more complicated.

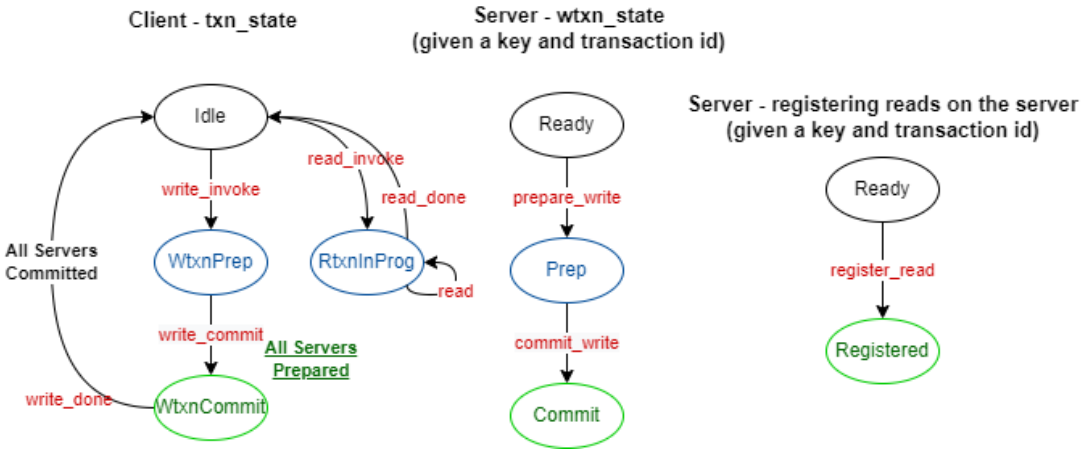


Figure 5.4: Eiger-PORT+ state diagrams, as modeled in Isabelle/HOL

## 5.4 Events

This section introduces the system events for read-only and write-only transactions. We first give the formal definitions of the events and describe them afterward.

There are a few helper functions used to define the events. Most of these are used by one event and explained in that event's description. The function ***tid_match s t*** is a predicate that appears in all system events, and checks whether the transaction *t*'s sequence number is equal

to the value of the *txn_sn* variable of its owner client (retrieved by *get_cl_txn t*) and is used to ensure the transaction is the active transaction of the client. Also, the function *get_txn_cl cl* retrieves the client's *cl*'s active transaction in the current state.

## Read-only transactions

### Client Events

- **read_invoke (cl, keys, s, s'):**
  $txn\_state(cl) = \textbf{\textit{Idle}} \triangleright txn\_state(cl) := \textbf{\textit{RtxnInProg}}\ keys\ (Map.empty) \ \wedge$
  $gst(cl) := \textbf{\textit{max}}\ (gst(cl))(\textbf{\textit{Min}}\ (\textbf{\textit{range}}\ (lst\_map(cl))))$

- **read (cl, k, v, s, s'):**
  $\exists keys\ vals.\ txn\_state(cl) = \textbf{\textit{RtxnInProg}}\ keys\ vals \ \wedge \ k \in keys \ \wedge \ vals\ k = \textbf{\textit{None}} \ \wedge$
  $\textbf{\textit{Some}}\ v = \textbf{\textit{find\_and\_read\_val}}\ (get\_txn\_cl\ cl)\ (DS(k)) \triangleright$
  $txn\_state(cl) := \textbf{\textit{RtxnInProg}}\ keys\ (vals\ (k \mapsto v)) \ \wedge \ lst\_map(cl)\ k := lst(k)$

- **read_done (cl, kv_map, sn, u", s, s'):**
  $txn\_state(cl) = \textbf{\textit{RtxnInProg}}\ (\textbf{\textit{dom}}\ kv\_map)\ kv\_map \ \wedge \ sn = txn\_sn(cl) \ \wedge$
  $u'' = cl\_view(s'\ cl) \triangleright txn\_state(cl) := \textbf{\textit{Idle}} \ \wedge \ txn\_sn(cl) := \textbf{\textit{Suc}}\ (txn\_sn(cl)) \ \wedge$
  $(\forall k \in \textbf{\textit{dom}}\ kv\_map.\ cl\_view(cl)\ k := \textbf{\textit{insert}}\ (ver\_id\ (\textbf{\textit{read\_at}}\ (DS(k))\ (gst(cl))\ cl))\ (cl\_view(cl)\ k))$

### Server Event

- **register_read (svr, t, v, i, gst_ts, s, s'):**
  $\textbf{\textit{tid\_match}}\ s\ t \wedge (\!|\ ver\_val = v,\ ver\_id = i\ |\!) = \textbf{\textit{read\_at}}\ (DS(svr))\ gst\_ts\ (get\_cl\_txn\ t) \wedge$
  $(\exists keys\ vals.\ txn\_state(get\_cl\_txn\ t) = \textbf{\textit{RtxnInProg}}\ keys\ vals \ \wedge \ svr \in keys \ \wedge$
  $vals\ svr = \textbf{\textit{None}}) \ \wedge \ gst\_ts = gst(get\_cl\_txn\ t) \triangleright clock(svr) := \textbf{\textit{Suc}}\ (clock(svr)) \ \wedge$
  $DS(svr) := \textbf{\textit{add\_to\_readerset}}\ (DS(svr))\ t\ i$

The sequence of events for executing a read-only transaction is as follows:

The client event *read_invoke* starts the read-only transaction and stores the keys to be read in the transaction state (*txn_state*) of the client alongside an empty map that will be filled with the read values, eventually. The servers handling the keys that are involved in the transaction will then register the reads (*register_read*) in the read version's reader sets. A *read* event for each registered read will then record this read in the transaction state of the client. Once all the keys have a value recorded for them, the *read_done* event can be executed. This is the event that simulates the abstract commit event and all the previous ones refine Skip, as their effect will not appear on the reconstructed abstract state (key-value store and client views) because we always remove pending transactions in the reconstruction.

The function $\boldsymbol{find\_and\_read\_val(t, vl)}$ searches for the appearance of the transaction $t$ in the reader-sets of the version list $vl$, used in the *read* event to check if $t$'s read has been registered, and if this is the case, then it returns the value of that version. The function $\boldsymbol{read\_at(vl, ts, cl)}$, used in the *register_read* and *read_done* events, first retrieves from the version list $vl$, a version $ver$ with the largest commit timestamp before the timestamp $ts$. Next, it checks whether there exists any newer version than $ver$ (with a higher commit timestamp) written by the client $cl$. If so, it returns the value and index of that version. Otherwise, it returns the value and index of the initially retrieved version $ver$. The function $\boldsymbol{add\_to\_readerset(vl, t, i)}$ adds (registers) transaction $t$ to the reader-set of version $i$ in the version list $vl$ in the *register_read* event.

## Write-only transactions

### Client Events

- **write_invoke (cl, kv_map, s, s'):**
  $txn\_state(cl) = \textbf{\textit{Idle}} \ \wedge \ kv\_map \neq Map.empty \ \triangleright \ txn\_state(cl) := \textbf{\textit{WtxnPrep}} \ kv\_map$

- **write_commit (cl, kv_map, commit_t, sn, u'', s, s'):**
  $txn\_state(cl) = \textbf{\textit{WtxnPrep}} \ kv\_map \ \wedge \ sn = txn\_sn(cl) \ \wedge \ u'' = cl\_view(cl) \ \wedge$
  $(\forall k \in \textbf{\textit{dom}} \ kv\_map. \ \exists i, prep\_t. \ wtxn\_state(k) \ (get\_txn\_cl \ cl) = \textbf{\textit{Prep}} \ prep\_t \ i) \ \wedge$
  $commit\_t = \textbf{\textit{Max}}\{prep\_t. \ (\exists i, k \in \textbf{\textit{dom}} \ kv\_map. \ wtxn\_state(k) \ (get\_txn\_cl \ cl) = \textbf{\textit{Prep}} \ prep\_t \ i)\} \ \triangleright$
  $txn\_state(cl) := \textbf{\textit{WtxnCommit}} \ commit\_t \ kv\_map \ \wedge$
  $(\forall k \in \textbf{\textit{dom}} \ kv\_map. \ cl\_view(cl) \ k := \textbf{\textit{insert}} \ (SOME \ i. \ \exists prep\_t. \ wtxn\_state(k) \ (get\_txn\_cl \ cl) =$
  $\textbf{\textit{Prep}} \ prep\_t \ i) \ (cl\_view(cl) \ k))$

- **write_done (cl, s, s'):**
  $\exists kv\_map \ commit\_t. \ txn\_state(cl) = \textbf{\textit{WtxnCommit}} \ commit\_t \ kv\_map \ \wedge$
  $(\forall k \in \textbf{\textit{dom}} \ kv\_map. \ wtxn\_state(k) \ (get\_txn\_cl \ cl) = \textbf{\textit{Commit}}) \ \triangleright$
  $txn\_state(cl) := \textbf{\textit{Idle}} \ \wedge \ txn\_sn(cl) := \textbf{\textit{Suc}} \ (txn\_sn(cl)) \ \wedge$
  $lst\_map(cl) := (\lambda k. \ \textbf{\textit{if}} \ k \in \textbf{\textit{dom}} \ kv\_map \ \textbf{\textit{then}} \ lst(k) \ \textbf{\textit{else}} \ lst\_map(cl))$

### Server Events

- **prepare_write (svr, t, v, gst_ts, s, s'):**
  $\textbf{\textit{tid\_match}} \ s \ t \ \wedge \ wtxn\_state(svr) \ t = \textbf{\textit{Ready}} \ \wedge$
  $(\exists kv\_map. \ txn\_state(get\_cl\_txn \ t) = \textbf{\textit{WtxnPrep}} \ kv\_map \ \wedge \ svr \in \textbf{\textit{dom}} \ kv\_map \ \wedge$
  $kv\_map \ svr = \textbf{\textit{Some}} \ v) \ \wedge \ gst\_ts = gst(get\_cl\_txn \ t) \ \triangleright$
  $wtxn\_state(svr) \ t := \textbf{\textit{Prep}} \ (\textbf{\textit{Suc}} \ (clock(svr))) \ (\textbf{\textit{length}} \ (DS(svr))) \ \wedge$
  $clock(svr) := \textbf{\textit{Suc}} \ (clock(svr)) \ \wedge \ pending\_wtxns(svr) \ t := \textbf{\textit{Some}} \ (clock(svr)) \ \wedge$
  $DS(svr) := DS(svr) \ @ \ [ \ (\!| \ v\_value = v, \ v\_writer = \textbf{\textit{Tn}} \ t, \ v\_readerset = \{\},$
  $v\_ts = clock(svr), \ v\_gst = gst\_ts, \ v\_is\_pending = \textbf{\textit{True}} \ |\!) \ ]$

- **commit_write (svr, t, s, s'):**
  ***tid_match*** $s\ t\ \wedge\ (\exists prep\_t\ i.\ wtxn\_state(svr)\ t = \textbf{\textit{Prep}}\ prep\_t\ i\ \wedge$
  $(\exists kv\_map\ \ commit\_t.\ \ txn\_state(get\_cl\_txn\ \ t)\ =\ \textbf{\textit{WtxnCommit}}\ \ commit\_t\ \ kv\_map\ \ \wedge$
  $svr \in \textbf{\textit{dom}}\ kv\_map\ \triangleright$
  $wtxn\_state(svr)\ t := \textbf{\textit{Commit}}\ \wedge\ pending\_wtxns(svr)\ t := \textbf{\textit{None}}\ \wedge$
  $lst(svr) := (\textbf{\textit{if dom}}\ (pending\_wtxns(s'svr)) = \{\}\ \textbf{\textit{then}}\ clock(svr)$
  $\textbf{\textit{else Min}}\ (\textbf{\textit{ran}}\ (pending\_wtxns(s'svr))))\ \wedge$
  $DS(svr) := \textbf{\textit{commit\_in\_vl}}\ (DS(svr))\ commit\_t\ i))$

The sequence of events for executing a write-only transaction is as follows:

The client event *write_invoke* starts the write-only transaction and stores the map of key-value pairs in the transaction state (*txn_state*) of the client. This is a signal for the involved servers to go into the prepared state, so *prepare_write* on servers adds a pending version to the version list of the respective keys and stores a prepared timestamp in the write transaction state (*wtxn_state*) of that server. After all involved servers have transitioned to the prepared state, the *write_commit* client event is triggered, and changes the transaction state to the committed state. This event simulates the abstract commit event, as at this point we have all the newly written versions in the key-value store so we can reconstruct the abstract state using that information. The *commit_write* events will then eventually follow in each server and commit the pending versions with the commit timestamp calculated by getting the maximum of prepared timestamps of the transaction. Finally, the *write_done* event marks the end of the write-only transaction. All the events except for the *write_commit* event refine Skip and we reconstruct the state by filtering out the versions that are pending while the transaction state is in the prepared state. As soon as the transaction state indicates commit, we count the corresponding pending versions written by that transaction in the reconstructed key-value store as permanent, thus simulating the effect of the abstract commit.

The function ***commit_in_vl(vl, ts, i)*** is used in the *commit_write* event to set the value of *v_is_pending* variable of the version $i$ in the version list $vl$ to *False* and add $ts$ as its commit timestamp.

## 5.5 Refinement

To verify that the Eiger-PORT+ protocol satisfies CC+, we use refinement. The abstract model is an execution test parameterized with $R_{ET}$ and $vShift_{ET}$ instantiated to CC from Table 3.1. So $R_{ET}$ is instantiated with $R_{CC} = SO \cup (\bigcup_k WR_{\mathcal{K}}\ (k))$, where $\mathcal{K}$ is the KVS, and $vShift_{ET}$ is instantiated with $vShift_{CC} = u \sqsubseteq u' \wedge (\forall t \in \mathcal{K}' \backslash \mathcal{K}.\ \forall k,\ i.\ (v\_writer(\mathcal{K}'(k,\ i)),\ t) \in SO^{?} \Rightarrow i \in u'(k))$, where $\mathcal{K}, u, \mathcal{K}'$, and $u'$ represent the KVS and the client view before and after the commit (See Section 3.1 for definitions).

### Simulation and mediator functions

**Simulation Function:** The abstract state is a *config*, containing the global KVS and the clients' views. The simulation function is defined with the help of the two following functions that reconstruct these two parts:

- ***kvs_of_gs*:** This function builds the global KVS of the abstract state by combining the version lists of all keys that are stored on KMs. It removes registered reads of the pending read transactions, and filters out the pending writes, i.e., versions written by a pending write transaction, unless the *txn_state* variable of the client indicates committed, in which case pending writes of that transaction are treated as committed versions and not filtered out. The filtering of pending reads and writes is done by the functions shown in Table 5.1.

| Functions | Effect in the abstract state reconstruction process |
|---|---|
| ***pending_rtxn*(*s*, *t*)** | A predicate indicating if $t$ is a **pending read transaction**, based on its owner client's *txn_state* and *txn_sn*. |
| ***get_ver_committed_rd*(*s*, *v*)** | Projects away extra fields of the version $v$ and removes all **pending read transactions** from $v$'s reader-set using ***pending_rtxn*(*s*, *t*)**. |
| ***pending_wtxn*(*s*, *t*)** | A predicate indicating if $t$ is a **pending write transaction**, based on its owner client's *txn_state* and *txn_sn*. |
| ***get_vl_committed_wr*(*s*, *vl*)** | Filters out from the version list $vl$, all the versions with a True ***v_is_pending*** and a **pending writer transaction** according to ***pending_wtxn*(*s*, *t*)**. |

Table 5.1: Functions used in the refinement's Simulation function

  $kvs\_of\_gs(s)$ first calls *get_vl_committed_wr* on all keys' version lists to filter out the pending versions and then constructs the final result by applying *get_ver_committed_rd* to all the remaining versions in version lists so that the extra version attributes, *v_is_pending* and *v_ts*, are removed, and pending read transactions are removed from their reader-sets.

- ***views_of_gs*:** This function builds the client views of the abstract state. It combines the *cl_view* variable from all client states. The *cl_view* variable keeps track of the client's view during the protocol and is only updated in *read_done* and *write_commit* events, which add the versions that were read or written by the client to the view. This ensures that at each point of the protocol, *cl_view* represents the expected view of the client from the abstract model's perspective.

**Mediator Function:** For read-only transactions, the *read_done* event simulates the abstract commit, because only at this point we can be sure that all reads of the transaction have been registered. Any other read-related event refines Skip, and already registered reads of the pending transaction are removed from the reconstructed state until the *read_done* event, where

all registered reads will appear in the reconstructed state at once to simulate the abstract commit. The fingerprint argument of the abstract commit event is built from the *kv_map* argument of *read_done*, and it only contains read operations for keys.

For write-only transactions, the *write_commit* event simulates the abstract commit. This is similar to what we had in the 2PL+2PC protocol refinement in Chapter 4, as write transactions in Eiger-PORT also use 2PC. The writes of a transaction are committed in the reconstructed state as soon as the transaction manager (here the client) indicates a committed state. The fingerprint argument of the abstract commit event is again built from the *kv_map* argument of *write_commit*, and it only contains write operations for keys.

### The refinement proof

For the refinement proof, we need to show that all effects of a transaction (in KVS and client views) are only visible once the client has finished a read-only transaction or has entered the *WtxnCommit* state for a write-only transaction.

Write-only transactions use the 2PC protocol, so the proof we have already given in Chapter A can be reused with a few modifications and adaptations.

Read-only transactions are more complicated, especially concerning view refinement. The committing event for read-only transactions is the *read_done* event, and none of the other read events should cause a change in the reconstructed abstract state, so we must make sure that none of the reads that are registered in the KVS are visible in the reconstructed abstract global KVS and client views before the transaction commits. Moreover, we need to show that the $fingerprint\_property$ (defined in Chapter 3) holds for the reads, i.e., the read versions correspond to the newest versions of each key in the client's view. This is true because the reading timestamp ($gst$) is monotonically increasing, and reading at a timestamp always either returns the last committed version before the timestamp or newer versions written by the client; so it never returns older versions. If a newer version (beyond the stable frontier) is read to preserve RYW, the same version will be returned every time the client tries to reread that key until its read timestamp grows larger than that version's commit time.

The properties and the system behavior that we informally described so far are formalized by numerous invariants and lemmas that make the refinement proof possible. These include but are not limited to the following categories:

- **Monotonicity lemmas:** This group of invariants are concerned with the monotonicity of different state variables. Lamport clocks increase monotonically; as a result, the local safe time of servers also monotonically increases as it depends on the clock, and always is either updated to the minimum of pending write transactions' timestamps, which always increase for new transactions, or to the current clock of the server if there are no pending transactions. Following the monotonicity of servers' $lst$ variables, the $lst\_map$ of clients

on each server identifier is also monotonic because it follows the updates of servers' $lst$ variables with a delay. Finally, as a result, the global safe time of each client ($gst$) is monotonically increasing because it is always updated to the maximum of its own value and minimum $lst$ of $lst\_map$.

- **Inequality invariant of the gst, lst_map, lsts and the clock:** Through these group of invariants, we prove for all clients of all reachable states, the following inequality holds: $\forall svr.\ gst(cl) < lst\_map[svr](cl) < lst(svr) < clock(svr)$.

- **KVS invariants:** These invariants are very similar to KVS invariants of our previous proof in Chapter 4, since they depend on the key-value store modeled similarly. Invariant *KVSNonEmp* shows that the version lists of all servers ($DS$ variable) are always non-empty, and this is true because they are all initialized with a $v_0$ version and only grow. Invariant *KVSSNonEmp* shows that the reconstructed KVS in the simulation function is also always non-empty as it is built based on the version lists and contains at least the $v_0$ committed versions. Invariant *KVSNotAllPending* states that there is always a committed version in a key's version list because the initial state contains a $v_0$ version for each key, which is not pending, and versions always change from pending to committed but never the opposite direction.

- **System state for past and future write TIDs:** These invariants are needed to show the expected write transaction status ($wtxn\_state$) of the transactions that were running in the past (have a smaller sequence number than the current transaction, stored in $txn\_sn$), or will be executed in future (larger sequence number than $txn\_sn$). For past write transactions, the status is always *Commit* since they always finish with a commit. For future write transactions, the status is *Ready*, as they have not been started.

A detailed list of all invariants and important lemmas of the refinement proof is given in Appendix B.

# Chapter 6

# Related Work

In this chapter, we discuss the related work, categorized into two groups of studies, which use *declarative* or *operational* semantics for formalizing consistency properties.

## Declarative specifications for consistency models

**Abstract executions.** Cerone et al. propose a framework for transactional consistency models with atomic visibility in [9]. The baseline consistency model in this framework is the Read Atomicity (RA), introduced in [1]; thus, all six consistency models they specify guarantee atomic visibility. Their specification of consistency models is declarative and based on *abstract executions*, which consists of a history (trace) of transactions and the visibility and arbitration relations between the transactions of that history. For the specification of each consistency model, a set of *consistency axioms* constrains the execution by allowing only certain histories. Essentially, the formalization uses *anomalies* in the history, such as fractured reads, lost update, or write skew, and declares the expected behavior by disallowing those anomalies.

Burckhardt et al. propose the *eventual consistency* model in [26] and introduce visibility and arbitration relations for transactions ordering to specify their model declaratively. The *abstract executions* method is not explicitly mentioned in this work, as it was later named as such by Cerone et al. work [9], but essentially they have similar semantics for specification of a consistency model.

**Dependency graphs.** This method was first introduced by Bernstein et al. in [27] for formalizing serializability. It was later adapted to weaker isolation levels by Adya's PhD thesis [8], where he defines dependency relations between reads and writes (WR, WW, and RW) and proposes dependency graphs based on those relations, with nodes as committed transactions and the edges as the dependencies. He uses this approach to specify different isolation levels of distributed systems.

Cerone et al. in [28] propose a set of algebraic laws (inequalities) for relating the two mentioned styles of specifications, i.e., abstract executions and dependency graphs. These laws can prove that a given client program does not violate consistency guarantees by avoiding anomalies.

In contrast to the algorithms mentioned in this group, our mechanized framework introduces an operational specification of consistency models. It also provides the possibility to specify and model distributed transactional protocols for verification. Furthermore, this verification is invariant-based and works with theorem proving. Instead of using traces, the framework uses the system state, the client views, and dependency relations on transactions visible to the committing client to verify that consistency guarantees are satisfied.

### Operational specifications for consistency models

Crooks et al. in [29] introduce a state-based formalization of isolation guarantees. They use Adya's work [8] as a baseline and focus on ANSI/SQL isolation levels [4]. They use *read states* and *commit tests* notions that are similar to client views and execution tests of our operational framework, with the difference that their approach depends on the set of all transactions (the complete trace) while our framework only uses the current state and poses restrictions on the visible transactions of the committing client.

Kaki et al. in [30] propose a program logic for compositional reasoning about weakly-isolated concurrent SQL transactions in a single-version key-value store. They develop an operational semantics and a proof system for relating application invariants to the database state, parameterized by weak isolation semantics. They also introduce an inference algorithm for identifying the weakest isolation level that guarantees a given set of consistency requirements. In contrast, our framework finds the strongest isolation level a protocol satisfies and identifies the strongest consistency guarantees for the given application.

Nagar and Jagannathan [31] propose an operational semantics based on abstract executions for detecting robustness violations in distributed transactions under a consistency model specification. They combine abstract executions and a dependency graph-based characterization of serializability and reduce the satisfiability violation identification problem to an SAT problem, and use SMT-solvers to discover anomalies in database programs.

S. Liu et al. [32] introduce a framework in Maude for formal specification of distributed transaction systems and consistency models. They specify nine common data consistency models using this framework and provide automation for analyzing formally specified distributed transaction systems in Maude, by automatically recording relevant history, and using *model checking* on the history to check for violations of consistency properties.

Xiong et al. [10] introduce an operational framework for specifying consistency models. We have formalized this framework in Isabelle/HOL and proved its assumptions as invariants of our model. Our mechanized framework is suitable for modeling and verifying distributed transactional database protocols against consistency models. Moreover, we have used refinement and proof techniques within a theorem-proving environment for the correctness proofs of protocols (fully *mechanized* framework), an unprecedented approach for rigorously verifying a database design's isolation guarantees.

# Chapter 7

# Conclusions

In this thesis, we formalized in Isabelle/HOL a framework of consistency models for verifying protocols and client programs. Using this framework, we modeled and verified a two-phase locking protocol combined with a two-phase commit protocol for a distributed database transaction system. We proved that this protocol satisfies serializability by refinement of an abstract execution test model, which we formalized in Isabelle/HOL for this purpose.

We also modeled the Eiger-PORT protocol, a state-of-the-art protocol for read-only and write-only transactions that satisfies causal consistency. We modified and optimized this protocol to satisfy a stronger consistency model, namely the causal+ consistency or causal convergence model, which additionally guarantees a total order of versions. To verify this in our framework, we defined the appropriate simulation function (state mappings) and mediator function (event mappings) and modeled it in Isabelle/HOL. The refinement proof of this protocol is in progress, but we have already proven numerous invariants about different characteristics of the protocol.

To our knowledge, this is the first mechanized verification framework for consistency properties of database transactions and the first general and fully mechanized correctness proof of a concurrency control protocol in such a framework. In particular, compared to existing works, there are two major differences:

1. Although formalizing consistency properties is not new, there is no mechanized, systematic way to rigorously verify a database design's isolation guarantees.

2. Many works have "mechanized" the consistency properties, but they either only work with testing or model checking or are tailored to specific application programs and lack a good coverage of the spectrum of properties.

The results of this study provide a foundation for further theoretical work, such as framework generalizations and modifications, and also serve as a basis for further practical verification work, applicable to both protocols and client programs.

## 7.1 Future Work

During the course of this thesis, we have proposed many new ideas for future projects. Some of these can be considered as a continuation of this thesis, and some are related but independent project opportunities. We list these suggestions in the order of their appeal in our opinion as follows:

- Finishing the verification proof of the optimized Eiger-PORT+ protocol and analyzing the resulting performance improvement by implementing the modified code, starting from the Eiger-PORT code base (available at [25]), and comparing their performance.
- Developing a stronger and more generic operational framework of consistency models by relaxing some restrictions established by the framework of Xiong et al. [10]. This can, for example, be done by:
  1. Relaxing the snapshot property to deal with:
     (a) Weaker consistency properties such as Monotonic Atomic View (MAV) [13] and Read Committed (RC)
     (b) Recent transaction algorithms such as RAMP [1] or LORA [33], which allow reads to fetch prepared-only versions.
  2. Relaxing the *last-write-wins* policy, to handle algorithms like RAMP-1PW [1].
- Refining the operational framework of consistency models to achieve a more generic (parameterized) distributed model.
- Continuing the refinement of the protocols in order to connect it to the Igloo framework and get further refinements into Igloo I/O specifications.

# Bibliography


[1] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Scalable atomic visibility with ramp transactions," *ACM Transactions on Database Systems (TODS)*, vol. 41, no. 3, pp. 1–45, 2016.

[2] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 401–416.

[3] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 385–400.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," *ACM SIGMOD Record*, vol. 24, no. 2, pp. 1–10, 1995.

[5] C. H. Papadimitriou, "The serializability of concurrent database updates," *Journal of the ACM (JACM)*, vol. 26, no. 4, pp. 631–653, 1979.

[6] R. Rajan, J. Boyle, A. Sastry, R. Cohen, D. Durham, and S. Herzog, "The COPS (Common Open Policy Service) Protocol," RFC 2748, Jan. 2000. [Online]. Available: https://www.rfc-editor.org/info/rfc2748

[7] H. Lu, S. Sen, and W. Lloyd, "Performance-Optimal Read-Only transactions," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 333–349. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/lu

[8] A. Adya, "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, 1999.

[9] A. Cerone, G. Bernardi, and A. Gotsman, "A framework for transactional consistency models with atomic visibility," in *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, ser. LIPIcs, L. Aceto and D. de Frutos-Escrig, Eds., vol. 42. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 58–71. [Online]. Available: https://doi.org/10.4230/LIPIcs.CONCUR.2015.58

[10] S. Xiong, A. Cerone, A. Raad, and P. Gardner, "Data consistency in transactional storage systems: A centralised semantics," in *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, ser. LIPIcs, R. Hirschfeld and T. Pape, Eds., vol. 166. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 21:1–21:31. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2020.21

[11] C. Sprenger, T. Klenze, M. Eilers, F. Wolf, P. Müller, M. Clochard, and D. Basin, "Igloo: Soundly linking compositional refinement and separation logic for distributed systems verification," in *ACM Program. Lang. 4, OOPSLA, Article 152*, 2020. [Online]. Available: https://doi.org/10.1145/3428220

[12] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *Acm Sigact News*, vol. 33, no. 2, pp. 51–59, 2002.

[13] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *Proc. VLDB Endow.*, vol. 7, no. 3, p. 181–192, nov 2013. [Online]. Available: https://doi.org/10.14778/2732232.2732237

[14] ANSI X3.135-1992, *American National Standard for Information Systems — Database Language — SQL (includes ANSI X3.168-1989).* November, 1992.

[15] S. Xiong, "Parametric operational semantics for consistency models," Ph.D. dissertation, Imperial College London, 2019.

[16] M. Van Steen and A. S. Tanenbaum, *Distributed systems.* Maarten van Steen Leiden, The Netherlands, 2017.

[17] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, p. 558–565, jul 1978. [Online]. Available: https://doi.org/10.1145/359545.359563

[18] B. Liskov and R. Ladin, "Highly available distributed services and fault-tolerant distributed garbage collection," in *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, 1986, pp. 29–39.

[19] T. Nipkow, M. Wenzel, and L. C. Paulson, Eds., *Isabelle/HOL - A Proof Assistant for Higher-Order Logic.* Springer Berlin Heidelberg, 2002. [Online]. Available: https://doi.org/10.1007%2F3-540-45949-9

[20] T. Nipkow and G. Klein, *Concrete semantics: with Isabelle/HOL.* Springer, 2014. [Online]. Available: https://doi.org/10.1007%2F978-3-319-10542-0

[21] E. W. Weisstein, ""invariant." from mathworld–a wolfram web resource," https://mathworld.wolfram.com/Invariant.html.

[22] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for Low-Latency Geo-Replicated storage," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13).* Lombard, IL: USENIX Association, Apr. 2013, pp. 313–328. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/lloyd

[23] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, "Cure: Strong semantics meets high availability and low latency," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS).* IEEE, 2016, pp. 405–414.

[24] H. Ouyang, H. Wei, and Y. Huang, "Checking causal consistency of mongodb," in *Proceedings of the 12th Asia-Pacific Symposium on Internetware*, 2020, pp. 209–216.

[25] P. S. Group, "Eiger-port," https://github.com/princeton-sns/Eiger-PORT, 2020.

[26] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv, "Eventually consistent transactions," in *Programming Languages and Systems*, H. Seidl, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 67–86.

[27] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys (CSUR)*, vol. 13, no. 2, pp. 185–221, 1981.

[28] A. Cerone, A. Gotsman, and H. Yang, "Algebraic Laws for Weak Consistency," in *28th International Conference on Concurrency Theory (CONCUR 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), R. Meyer and U. Nestmann, Eds., vol. 85. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 26:1–26:18. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/7794

[29] N. Crooks, Y. Pu, L. Alvisi, and A. Clement, "Seeing is believing: A client-centric specification of database isolation," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, ser. PODC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 73–82. [Online]. Available: https://doi.org/10.1145/3087801.3087802

[30] G. Kaki, K. Nagar, M. Najafzadeh, and S. Jagannathan, "Alone together: Compositional reasoning and inference for weak isolation," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, dec 2017. [Online]. Available: https://doi.org/10.1145/3158115

[31] K. Nagar and S. Jagannathan, "Automated Detection of Serializability Violations Under Weak Consistency," in *29th International Conference on Concurrency Theory (CONCUR 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Schewe and L. Zhang, Eds., vol. 118. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 41:1–41:18. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2018/9579

[32] S. Liu, P. C. Ölveczky, M. Zhang, Q. Wang, and J. Meseguer, "Automatic analysis of consistency properties of distributed transaction systems in maude," in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Vojnar and L. Zhang, Eds. Cham: Springer International Publishing, 2019, pp. 40–57.

[33] S. Liu, "All in one: Design, verification, and implementation of snow-optimal read atomic transactions," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–44, 2022.

# Appendices

# Appendix A

# 1st Verification - List of Invariants

Isabelle theories for the framework and protocol verification proofs can be found at Distributed Database Transactions private repository available at https://gitlab.inf.ethz.ch/PRV-BASIN/christoph/distributed-database-transactions.

On the following pages you can find a list of all invariants and some of the important lemmas used in the verification proof of Two-Phase Locking protocol discussed in Chapter 4.

```
section ‹2PL+2PC Refinement Proof Invariants (and important lemmas)›

theory Serializable_2PC_2PL_Invariants
  imports Serializable_2PC_2PL
begin

— ‹Invariant about future and past transactions kms›

definition TIDFutureKm where
  "TIDFutureKm s cl ⟷ (∀n k. n > tm_sn (tm s cl) ⟶ km_status (kms s k) (Tn_cl n cl) = working)"

definition TIDPastKm where
  "TIDPastKm s cl ⟷ (∀n k. n < tm_sn (tm s cl) ⟶ km_status (kms s k) (Tn_cl n cl) ∈ {committed, aborted})"

lemma other_sn_idle:
  assumes "TIDFutureKm s cl" and "TIDPastKm s cl"
    and "get_cl_txn t = cl" and "get_sn_txn t ≠ tm_sn (tm s cl)"
  shows "⋀k. km_status (kms s k) t ∈ {working, committed, aborted}"


— ‹Lock Invariants›

definition RLockInv where
  "RLockInv s k ⟷ (∀t. km_status (kms s k) t = read_lock ⟶ (∀t. km_status (kms s k) t ≠ write_lock))"

definition WLockInv where
  "WLockInv s k ⟷ (∀t. km_status (kms s k) t ≠ write_lock) ∨ (∃! t. km_status (kms s k) t = write_lock)"

— ‹Invariants for fingerprint, knowing the lock (km status)›

definition RLockFpInv where
  "RLockFpInv s k ⟷ (∀t. km_status (kms s k) t = read_lock ⟶
    km_key_fp (kms s k) t W = None ∧
    km_key_fp (kms s k) t R ≠ None)"

definition WLockFpInv where
  "WLockFpInv s k ⟷ (∀t. km_status (kms s k) t = write_lock ⟶ km_key_fp (kms s k) t W ≠ None)"

definition NoLockFpInv where
  "NoLockFpInv s k ⟷ (∀t. km_status (kms s k) t = no_lock ⟶
    km_key_fp (kms s k) t W = None ∧
    km_key_fp (kms s k) t R = None)"

— ‹Invariants about kv store›
definition KVSNonEmp where
  "KVSNonEmp s ⟷ (∀k. km_vl (kms s k) ≠ [])"

definition KVSGSNonEmp where
  "KVSGSNonEmp s ⟷ (∀k. kvs_of_gs s k ≠ [])"

definition KVSLen where
  "KVSLen s cl ⟷ (∀k. length (km_vl (kms s k)) ≤ length (kvs_of_gs s k))"

subsubsection ‹Lemmas for kvs_of_gs changing by different events›

lemma kvs_of_gs_km_inv:
  assumes "WLockInv s k" and "RLockInv s k"
    and "(∀t. km_status (kms s k) t ≠ write_lock) ∨
              km_status (kms s' k) t ≠ write_lock"
    and "tm_status (tm s (get_cl_txn t)) ≠ tm_committed"
    and "⋀k. km_vl (kms s' k) = km_vl (kms s k)"
    and "tm_km_k'_t'_unchanged k s s' t"
  shows "kvs_of_gs s' = kvs_of_gs s"


lemma kvs_of_gs_tm_inv:
  assumes "TIDFutureKm s cl" and "TIDPastKm s cl"
    and "tm_status (tm s cl) ≠ tm_committed ∨
         (∀k. km_status (kms s k) (get_txn_cl cl s) = committed)"
    and "tm_status (tm s' cl) ≠ tm_committed"
    and "km_tm_cl'_unchanged cl s s'"
  shows "kvs_of_gs s' = kvs_of_gs s"
```

```
lemma update_kv_all_tm_commit_no_lock_inv:
  assumes "TIDPastKm s cl" and "TIDFutureKm s cl"
    and "tm_status (tm s cl) = tm_prepared"
    and "tm_status (tm s' cl) = tm_committed"
    and "other_insts_unchanged cl (tm s) (tm s')"
    and "km_status (kms s k) (get_txn_cl cl s) = no_lock"
  shows "update_kv_all_txn (λt. tm_status (tm s' (get_cl_txn t))) (km_status (kms s k))
          (km_key_fp (kms s k)) (km_vl (kms s k)) =
         update_kv_all_txn (λt. tm_status (tm s (get_cl_txn t))) (km_status (kms s k))
          (km_key_fp (kms s k)) (km_vl (kms s k))"


(*All events*)
abbreviation not_tm_commit where
  "not_tm_commit e ≡ ∀cl sn u F. e ≠ TM_Commit cl sn u F"

abbreviation invariant_list_kvs where
  "invariant_list_kvs s ≡ ∀cl k. TIDFutureKm s cl ∧ TIDPastKm s cl ∧ RLockInv s k ∧ WLockInv s k ∧
                          RLockFpInv s k ∧ NoLockFpInv s k ∧ KVSNonEmp s"

lemma kvs_of_gs_inv:
  assumes "gs_trans s e s'"
    and "invariant_list_kvs s"
    and "not_tm_commit e"
  shows "kvs_of_gs s' = kvs_of_gs s"


— ‹More specific lemmas about TM commit›
lemma kvs_of_gs_commit_length_increasing:
  assumes "tm_status (tm s cl) = tm_prepared"
    and "tm_status (tm s' cl) = tm_committed"
    and "km_tm_cl'_unchanged cl s s'"
  shows "length (kvs_of_gs s k) ≤ length (kvs_of_gs s' k)"


lemma kvs_of_gs_length_increasing:
  assumes "gs_trans s e s'"
    and "invariant_list_kvs s"
  shows "⋀k. length (kvs_of_gs s k) ≤ length (kvs_of_gs s' k)"


— ‹Fingerprint content invariant and Lemmas for proving the fp_property›

lemma km_vl_read_lock_commit_eq_length:
  assumes "RLockFpInv s k"
    and "km_status (kms s k) t = read_lock"
    and "km_vl (kms s' k) =
          update_kv_key t (km_key_fp (kms s k) t) (full_view (km_vl (kms s k))) (km_vl (kms s k))"
  shows "length (km_vl (kms s' k)) = length (km_vl (kms s k))"


definition RLockFpContentInv where
  "RLockFpContentInv s k ⟷ (∀t. km_status (kms s k) t = read_lock ⟶
    km_key_fp (kms s k) t R =
      Some (v_value (last_version (km_vl (kms s k)) (full_view (km_vl (kms s k))))))"

definition WLockFpContentInv where
  "WLockFpContentInv s k ⟷ (∀t. km_status (kms s k) t = write_lock ⟶
    km_key_fp (kms s k) t R = None ∨
    km_key_fp (kms s k) t R =
      Some (v_value (last_version (km_vl (kms s k)) (full_view (km_vl (kms s k))))))"

lemma km_vl_kvs_eq_length:
  assumes "WLockInv s k" and "RLockInv s k"
    and "tm_status (tm s cl) = tm_prepared"
    and "km_status (kms s k) (get_txn_cl cl s) ∈ {read_lock, write_lock}"
  shows "length (kvs_of_gs s k) = length (km_vl (kms s k))"


— ‹Lemmas for view growth after commit›

lemma committed_kvs_view_grows:
  assumes "tm_status (tm s cl) = tm_prepared"
    and "tm_status (tm s' cl) = tm_committed"
```

```
    and "km_tm_cl'_unchanged cl s s'"
  shows "(λk. full_view (kvs_of_gs s k)) ⊑ (λk. full_view (kvs_of_gs s' k))"


lemma updated_vl_view_grows:
  assumes "km_vl (kms s' k) =
    update_kv_key t (km_key_fp (kms s k) t) (full_view (km_vl (kms s k))) (km_vl (kms s k))"
    and "other_insts_unchanged k (kms s) (kms s')"
  shows "(λk. full_view (km_vl (kms s k))) ⊑ (λk. full_view (km_vl (kms s' k)))"


lemma tm_view_inv:
  assumes "gs_trans s e s'"
    and "not_tm_commit e"
  shows "tm_view (tm s' cl) = tm_view (tm s cl)"


definition TMFullView where
  "TMFullView s cl ⟷ tm_view (tm s cl) ⊑ (λk. full_view (kvs_of_gs s k))"
```

— ‹TM_commit updating kv›

```
lemma kvs_of_gs_tm_commit:
  assumes "TIDFutureKm s cl" and "TIDPastKm s cl"
    and "WLockInv s k" and "WLockFpInv s k"
    and "RLockInv s k" and "RLockFpInv s k"
    and "NoLockFpInv s k" and "KVSNonEmp s"
    and "tm_status (tm s cl) = tm_prepared"
    and "km_status (kms s k) (get_txn_cl cl s) ∈ {read_lock, write_lock, no_lock}"
    and "tm_status (tm s' cl) = tm_committed"
    and "other_insts_unchanged cl (tm s) (tm s')"
  shows "update_kv_all_txn (λt. tm_status (tm s' (get_cl_txn t))) (km_status (kms s k))
          (km_key_fp (kms s k)) (km_vl (kms s k)) =
    update_kv_key (get_txn_cl cl s) (km_key_fp (kms s k) (get_txn_cl cl s))
      (full_view (update_kv_all_txn (λt. tm_status (tm s (get_cl_txn t))) (km_status (kms s k))
        (km_key_fp (kms s k)) (km_vl (kms s k))))
      (update_kv_all_txn (λt. tm_status (tm s (get_cl_txn t))) (km_status (kms s k))
        (km_key_fp (kms s k)) (km_vl (kms s k)))"
```

— ‹Lemmas for showing transaction id freshness›

```
lemma get_sqns_other_cl_inv:
  assumes "TIDFutureKm s cl" and "TIDPastKm s cl"
    and "⋀k. WLockInv s k" and "⋀k. WLockFpInv s k"
    and "⋀k. RLockInv s k" and "⋀k. RLockFpInv s k"
    and "⋀k. NoLockFpInv s k" and "KVSNonEmp s"
    and "tm_status (tm s cl) = tm_prepared"
    and "tm_status (tm s' cl) = tm_committed"
    and "⋀k. km_status (kms s k) (get_txn_cl cl s) = read_lock ∨
             km_status (kms s k) (get_txn_cl cl s) = write_lock ∨
             km_status (kms s k) (get_txn_cl cl s) = no_lock"
    and "km_tm_cl'_unchanged cl s s'"
    and "cl' ≠ cl"
  shows "get_sqns (kvs_of_gs s') cl' = get_sqns (kvs_of_gs s) cl'"


lemma new_t_is_in_writers:
  assumes "TIDFutureKm s cl" and "TIDPastKm s cl"
    and "⋀k. WLockInv s k" and "⋀k. WLockFpInv s k"
    and "⋀k. RLockInv s k" and "⋀k. RLockFpInv s k"
    and "⋀k. NoLockFpInv s k" and "KVSNonEmp s"
    and "tm_status (tm s cl) = tm_prepared"
    and "tm_status (tm s' cl) = tm_committed"
    and "km_status (kms s k) (get_txn_cl cl s) = write_lock"
    and "other_insts_unchanged cl (tm s) (tm s')"
    and "kms s' = kms s"
  shows "vl_writers_sqns (kvs_of_gs s' k) cl = vl_writers_sqns (kvs_of_gs s k) cl ∪ {tm_sn (tm s cl)}"


lemma new_t_is_in_writers2:
  assumes "TIDFutureKm s cl" and "TIDPastKm s cl"
    and "⋀k. WLockInv s k" and "⋀k. WLockFpInv s k"
    and "⋀k. RLockInv s k" and "⋀k. RLockFpInv s k"
    and "⋀k. NoLockFpInv s k" and "KVSNonEmp s"
```

```
    and "tm_status (tm s cl) = tm_prepared"
    and "tm_status (tm s' cl) = tm_committed"
    and "km_status (kms s k) (get_txn_cl cl s) = read_lock"
    and "other_insts_unchanged cl (tm s) (tm s')"
    and "kms s' = kms s"
  shows "vl_writers_sqns (kvs_of_gs s' k) cl = vl_writers_sqns (kvs_of_gs s k) cl"


lemma new_t_is_in_readers:
  assumes "TIDFutureKm s cl" and "TIDPastKm s cl"
    and "⋀k. WLockInv s k" and "⋀k. WLockFpInv s k"
    and "⋀k. RLockInv s k" and "⋀k. RLockFpInv s k"
    and "⋀k. NoLockFpInv s k" and "KVSNonEmp s"
    and "tm_status (tm s cl) = tm_prepared"
    and "tm_status (tm s' cl) = tm_committed"
    and "km_status (kms s k) (get_txn_cl cl s) = read_lock"
    and "other_insts_unchanged cl (tm s) (tm s')"
    and "kms s' = kms s"
  shows "vl_readers_sqns (kvs_of_gs s' k) cl = vl_readers_sqns (kvs_of_gs s k) cl ∪ {tm_sn (tm s cl)}"


lemma new_t_is_in_readers2:
  assumes "TIDFutureKm s cl" and "TIDPastKm s cl"
    and "⋀k. WLockInv s k" and "⋀k. WLockFpInv s k"
    and "⋀k. RLockInv s k" and "⋀k. RLockFpInv s k"
    and "⋀k. NoLockFpInv s k" and "KVSNonEmp s"
    and "tm_status (tm s cl) = tm_prepared"
    and "tm_status (tm s' cl) = tm_committed"
    and "km_status (kms s k) (get_txn_cl cl s) = write_lock"
    and "km_key_fp (kms s k) (get_txn_cl cl s) R ≠ None"
    and "other_insts_unchanged cl (tm s) (tm s')"
    and "kms s' = kms s"
  shows "vl_readers_sqns (kvs_of_gs s' k) cl = vl_readers_sqns (kvs_of_gs s k) cl ∪ {tm_sn (tm s cl)}"


lemma new_t_is_in_readers3:
  assumes "TIDFutureKm s cl" and "TIDPastKm s cl"
    and "⋀k. WLockInv s k" and "⋀k. WLockFpInv s k"
    and "⋀k. RLockInv s k" and "⋀k. RLockFpInv s k"
    and "⋀k. NoLockFpInv s k" and "KVSNonEmp s"
    and "tm_status (tm s cl) = tm_prepared"
    and "tm_status (tm s' cl) = tm_committed"
    and "km_status (kms s k) (get_txn_cl cl s) = write_lock"
    and "km_key_fp (kms s k) (get_txn_cl cl s) R = None"
    and "other_insts_unchanged cl (tm s) (tm s')"
    and "kms s' = kms s"
  shows "vl_readers_sqns (kvs_of_gs s' k) cl = vl_readers_sqns (kvs_of_gs s k) cl"


lemma kvs_writers_tm_commit_grows:
  assumes "TIDFutureKm s cl" and "TIDPastKm s cl"
    and "⋀k. WLockInv s k" and "⋀k. WLockFpInv s k"
    and "⋀k. RLockInv s k" and "⋀k. RLockFpInv s k"
    and "⋀k. NoLockFpInv s k" and "KVSNonEmp s"
    and "tm_status (tm s cl) = tm_prepared"
    and "tm_status (tm s' cl) = tm_committed"
    and "∀k. km_status (kms s k) (get_txn_cl cl s) ∈ {read_lock, write_lock, no_lock}"
    and "km_status (kms s k) (get_txn_cl cl s) = write_lock"
    and "other_insts_unchanged cl (tm s) (tm s')"
    and "kms s' = kms s"
  shows "kvs_writers_sqns (kvs_of_gs s') cl = kvs_writers_sqns (kvs_of_gs s) cl ∪ {tm_sn (tm s cl)}"


lemma kvs_writers_tm_commit_doesnt_grow:
  assumes "TIDFutureKm s cl" and "TIDPastKm s cl"
    and "⋀k. WLockInv s k" and "⋀k. WLockFpInv s k"
    and "⋀k. RLockInv s k" and "⋀k. RLockFpInv s k"
    and "⋀k. NoLockFpInv s k" and "KVSNonEmp s"
    and "tm_status (tm s cl) = tm_prepared"
    and "tm_status (tm s' cl) = tm_committed"
    and "∀k. km_status (kms s k) (get_txn_cl cl s) ∈ {read_lock, no_lock}"
    and "other_insts_unchanged cl (tm s) (tm s')"
    and "kms s' = kms s"
  shows "kvs_writers_sqns (kvs_of_gs s') cl = kvs_writers_sqns (kvs_of_gs s) cl"
```

```
lemma kvs_readers_sqns_tm_commit_grows:
  assumes "TIDFutureKm s cl" and "TIDPastKm s cl"
    and "⋀k. WLockInv s k" and "⋀k. WLockFpInv s k"
    and "⋀k. RLockInv s k" and "⋀k. RLockFpInv s k"
    and "⋀k. NoLockFpInv s k" and "KVSNonEmp s"
    and "tm_status (tm s cl) = tm_prepared"
    and "tm_status (tm s' cl) = tm_committed"
    and "∀k. km_status (kms s k) (get_txn_cl cl s) ∈ {read_lock, write_lock, no_lock}"
    and "km_status (kms s k) (get_txn_cl cl s) = read_lock ∨
         (km_status (kms s k) (get_txn_cl cl s) = write_lock ∧
          km_key_fp (kms s k) (get_txn_cl cl s) R ≠ None)"
    and "other_insts_unchanged cl (tm s) (tm s')"
    and "kms s' = kms s"
  shows "kvs_readers_sqns (kvs_of_gs s') cl = kvs_readers_sqns (kvs_of_gs s) cl ∪ {tm_sn (tm s cl)}"


lemma kvs_readers_sqns_tm_commit_doesnt_grow:
  assumes "TIDFutureKm s cl" and "TIDPastKm s cl"
    and "⋀k. WLockInv s k" and "⋀k. WLockFpInv s k"
    and "⋀k. RLockInv s k" and "⋀k. RLockFpInv s k"
    and "⋀k. NoLockFpInv s k" and "KVSNonEmp s"
    and "tm_status (tm s cl) = tm_prepared"
    and "tm_status (tm s' cl) = tm_committed"
    and "∀k. km_status (kms s k) (get_txn_cl cl s) ∈ {write_lock, no_lock}"
    and "∀k. km_status (kms s k) (get_txn_cl cl s) ≠ write_lock ∨
              km_key_fp (kms s k) (get_txn_cl cl s) R = None"
    and "other_insts_unchanged cl (tm s) (tm s')"
    and "kms s' = kms s"
  shows "kvs_readers_sqns (kvs_of_gs s') cl = kvs_readers_sqns (kvs_of_gs s) cl"


lemma get_sqns_tm_commit_grows:
  assumes "TIDFutureKm s cl" and "TIDPastKm s cl"
    and "⋀k. WLockInv s k" and "⋀k. WLockFpInv s k"
    and "⋀k. RLockInv s k" and "⋀k. RLockFpInv s k"
    and "⋀k. NoLockFpInv s k" and "KVSNonEmp s"
    and "tm_status (tm s cl) = tm_prepared"
    and "tm_status (tm s' cl) = tm_committed"
    and "⋀k. km_status (kms s k) (get_txn_cl cl s) ∈ {read_lock, write_lock, no_lock}"
    and "other_insts_unchanged cl (tm s) (tm s')"
    and "kms s' = kms s"
  shows "get_sqns (kvs_of_gs s') cl =
         (if ∀k. km_status (kms s k) (get_txn_cl cl s) = no_lock then
          get_sqns (kvs_of_gs s) cl else
          get_sqns (kvs_of_gs s) cl ∪ {tm_sn (tm s cl)})"


definition SqnInv where
  "SqnInv s cl ⟷
    (tm_status (tm s cl) ≠ tm_committed ⟶ (∀m ∈ get_sqns (kvs_of_gs s) cl. m < tm_sn (tm s cl))) ∧
    (tm_status (tm s cl) = tm_committed ⟶ (∀m ∈ get_sqns (kvs_of_gs s) cl. m ≤ tm_sn (tm s cl)))"


— ‹Lemmas for proving view wellformedness of tm_view›

lemma kvs_of_gs_version_order:
  assumes "TIDPastKm s cl" and "TIDFutureKm s cl" and "WLockInv s k" and "RLockInv s k" and "KVSNonEmp s"
    and "i ∈ full_view (kvs_of_gs s k)"
    and "tm_status (tm s cl) = tm_prepared"
    and "tm_status (tm s' cl) = tm_committed"
    and "km_status (kms s k) (get_txn_cl cl s) ∈ {read_lock, write_lock, no_lock}"
    and "km_tm_cl'_unchanged cl s s'"
  shows "kvs_of_gs s k ! i ⊑ver kvs_of_gs s' k ! i"


lemma new_version_index:
  assumes "TIDPastKm s cl" and "TIDFutureKm s cl"
    and "WLockInv s k" and "KVSNonEmp s"
    and "tm_status (tm s cl) = tm_prepared"
    and "tm_status (tm s' cl) = tm_committed"
    and "km_status (kms s k) (get_txn_cl cl s) = write_lock"
    and "other_insts_unchanged cl (tm s) (tm s')"
    and "i ∈ full_view (update_kv_all_txn (λt. tm_status (tm s' (get_cl_txn t)))
    (km_status (kms s k)) (km_key_fp (kms s k)) (km_vl (kms s k)))"
```

```
    and "i ∉ full_view (update_kv_all_txn (λt. tm_status (tm s (get_cl_txn t)))
    (km_status (kms s k)) (km_key_fp (kms s k)) (km_vl (kms s k)))"
  shows "i = length (km_vl (kms s k))"


lemma t_is_fresh:
  assumes "SqnInv s cl"
    and "tm_status (tm s cl) = tm_prepared"
  shows "get_txn_cl cl s ∈ next_txids (kvs_of_gs s) cl"


lemma kvs_of_gs_view_atomic:
  assumes "TIDPastKm s cl" and "TIDFutureKm s cl"
    and "⋀k. WLockInv s k" and "⋀k. WLockFpInv s k"
    and "⋀k. RLockInv s k" and "⋀k. NoLockFpInv s k"
    and "SqnInv s cl" and "KVSNonEmp s"
    and "tm_status (tm s cl) = tm_prepared"
    and "tm_status (tm s' cl) = tm_committed"
    and "∀k. km_status (kms s k) (get_txn_cl cl s) ∈ {read_lock, write_lock, no_lock}"
    and "km_tm_cl'_unchanged cl s s'"
  shows "view_atomic (kvs_of_gs s') (λk. full_view (kvs_of_gs s k))"


lemma reach_kvs_expands [simp, intro]:
  assumes "reach tps s" and "gs_trans s e s'"
    and "⋀cl. TIDFutureKm s cl" and "⋀cl. TIDPastKm s cl"
    and "⋀k. RLockInv s k" and "⋀k. WLockInv s k"
    and "⋀k. RLockFpInv s k" and "⋀k. NoLockFpInv s k"
    and "KVSNonEmp s" and "KVSLen s cl"
  shows "kvs_of_gs s ⊑kvs kvs_of_gs s'"


definition KVSView where
  "KVSView s cl ⟷ view_wellformed (kvs_of_gs s) (tm_view (tm s cl))"

— ‹CanCommit›

lemma writers_visible:
  assumes "u = (λk. full_view (K k))"
  shows "visTx K u = kvs_writers K"


lemma WW_writers_id_helper:
  assumes "(x, v_writer x') ∈ {(xa, x). ∃xb i.
            i ∈ full_view (K xb) ∧
            (∃i'. i' ∈ full_view (K xb) ∧
              x = v_writer (K xb ! i) ∧ xa = v_writer (K xb ! i') ∧ i < i')}* "
    and "x' ∈ set (K k)"
  shows "∃xa. x ∈ v_writer ` set (K xa)"


lemma WW_writers_id:
  "((⋃ (range (WW K)))⁻¹)*)⁻¹ `` kvs_writers K = kvs_writers K"


lemma full_view_satisfies_ET_SER_canCommit:
  "u = (λk. full_view (K k)) ⟹ ET_SER.canCommit K u F"


end
```

# Appendix B

# 2nd Verification - List of Invariants

On the following pages you can find a list of all invariants and some of the important lemmas used in the verification proof of Optimized Eiger-Port protocol discussed in Chapter 5.

```
section ‹Eiger Port+ Refinement Proof Invariants (and important lemmas)›

theory CCv_Eiger_Port_modified_Invariants
  imports CCv_Eiger_Port_modified
begin

— ‹Invariants about kv store›
definition KVSNonEmp where
  "KVSNonEmp s ⟷ (∀k. DS (svrs s k) ≠ [])"

definition KVSNotAllPending where
  "KVSNotAllPending s k ⟷ (∃i. i < length (DS (svrs s k)) ∧ ¬v_is_pending (DS (svrs s k) ! i))"

definition KVSSNonEmp where
  "KVSSNonEmp s ⟷ (∀k. kvs_of_s s k ≠ [])"

— ‹Invariant about future and past transactions svrs›

definition FutureTIDInv where
  "FutureTIDInv s cl ⟷ (∀n k. n > txn_sn (cls s cl) ⟶ wtxn_state (svrs s k) (Tn_cl n cl) = Ready)"

definition PastTIDInv where
  "PastTIDInv s cl ⟷ (∀n k. n < txn_sn (cls s cl) ⟶ wtxn_state (svrs s k) (Tn_cl n cl) ∈ {Ready, Commit})"

lemma other_sn_idle:
  assumes "FutureTIDInv s cl" and "PastTIDInv s cl"
    and "get_cl_txn t = cl" and "get_sn_txn t ≠ txn_sn (cls s cl)"
  shows "⋀k. wtxn_state (svrs s k) t ∈ {Ready, Commit}"


abbreviation not_committing_ev where
  "not_committing_ev e ≡ ∀cl kv_map cts sn u. e ≠ RDone cl kv_map sn u ∧ e ≠ WCommit cl kv_map cts sn u"

abbreviation invariant_list_kvs where
  "invariant_list_kvs s ≡ ∀cl k. FutureTIDInv s cl ∧ PastTIDInv s cl ∧ KVSNonEmp s ∧ KVSNotAllPending s k"

subsection ‹Refinement Proof›

lemma pending_rtxn_inv:
  assumes "∀keys kv_map. txn_state (cls s cl) ≠ RtxnInProg keys kv_map"
    and "∀keys kv_map. txn_state (cls s' cl) ≠ RtxnInProg keys kv_map"
    and "∀cl'. cl' ≠ cl ⟶ cls s' cl' = cls s cl'"
  shows "pending_rtxn s' t = pending_rtxn s t"


lemma pending_wtxn_inv:
  assumes "∀kv_map. txn_state (cls s cl) ≠ WtxnPrep kv_map"
    and "∀kv_map. txn_state (cls s' cl) ≠ WtxnPrep kv_map"
    and "∀cl'. cl' ≠ cl ⟶ cls s' cl' = cls s cl'"
  shows "pending_wtxn s' t = pending_wtxn s t"


lemma kvs_of_s_inv:
  assumes "state_trans s e s'"
    and "invariant_list_kvs s"
    and "not_committing_ev e"
  shows "kvs_of_s s' = kvs_of_s s"


lemma finite_pending_wtxns:
  assumes "pending_wtxns (svrs s' k) t = Some x"
    and "∀k'. finite (ran (pending_wtxns (svrs s k')))"
    and "∀k'. k' ≠ k ⟶ pending_wtxns (svrs s' k') = pending_wtxns (svrs s k')"
    and "∀t'. t' ≠ t ⟶ pending_wtxns (svrs s' k) t' = pending_wtxns (svrs s k) t'"
  shows "∀k. finite (ran (pending_wtxns (svrs s' k)))"


definition FinitePendingInv where
  "FinitePendingInv s svr ⟷ finite (ran (pending_wtxns (svrs s svr)))"

lemma clock_monotonic:
  assumes "state_trans s e s'"
  shows "clock (svrs s' svr) ≥ clock (svrs s svr)"


definition PendingWtsInv where
  "PendingWtsInv s ⟷ (∀svr. ∀ts ∈ ran (pending_wtxns (svrs s svr)). ts ≤ clock (svrs s svr))"

definition ClockLstInv where
  "ClockLstInv s ⟷ (∀svr. lst (svrs s svr) ≤ clock (svrs s svr))"
```

```
lemma lst_monotonic:
  assumes "state_trans s e s'"
  shows "lst (svrs s' svr) ≥ lst (svrs s svr)"


lemma gst_monotonic:
  assumes "state_trans s e s'"
  shows "gst (cls s' cl) ≥ gst (cls s cl)"


lemma tm_view_inv:
  assumes "state_trans s e s'"
    and "not_committing_ev e"
  shows "cl_view (cls s' cl) = cl_view (cls s cl)"


end
```

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

## Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

---

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

**Title of work** (in block letters):

Mechanized Data Consistency Models for Distributed Database Transactions

**Authored by** (in block letters):

*For papers written by groups the names of all authors are required.*

| **Name(s):** | **First name(s):** |
|---|---|
| Ghasemirad | Shabnam |
| | |
| | |
| | |
| | |

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

| **Place, date** | **Signature(s)** |
|---|---|
| Zürich, 03.11.2022 | Shabnam Ghasemirad |
| | |
| | |
| | |
| | |

*For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.*