Imperial College
London

MASTER'S THESIS

DEPARTMENT OF COMPUTING

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

Operational Semantics for Distributed Atomic Transactions: a 'Perfect' Understanding

Author:
Boaz Francis

Supervisor:
Prof. Philippa Gardner

Second Marker:
Dr. John Wickerson

June, 2020

Abstract

We provide a forward-looking reformulation of the interleaving centralised operational semantics for capturing the client-observable behaviour of distributed atomic transactions of Xiong et al. [Data Consistency in Transactional Storage Systems: a Centralised Approach, 2019], and prove its equivalence to the original formulation. Our formalism operates on *minimal views*, which capture the knowledge a client has of the key-value store immediately after committing a transaction. We focus on the causal consistency model because it allows us to think about client views without associating them with a particular transaction. Using our minimal views, we demonstrate a systematic way for extending client views to incorporate information from the environment, eliminating the backwards analysis required by the original semantics. We apply the concept of minimal views to snapshots and present an algorithm for producing all the obtainable stores given a client program under causal consistency.

We first evaluate the work of Xiong et al. from an implementation perspective and identify the challenging parts. We introduce our revised formalism and show that it is equivalent to the original. Using minimal views to alter the definition of snapshots, we suggest several approaches for extending client views in a forward way, focusing on the causal consistency model. Last, we present an algorithm for implementing our approach and suggest potential optimisations.

Acknowledgements

I would like to thank Shale Xiong for his insights and constant technical support throughout this project. Also, I thank John Wickerson and Petar Maksimović for their advice and suggestions in the early stages.

My biggest thanks go to my supervisor, Philippa Gardner, for her continuous inspiration and direction throughout the course of the project.

Contents

List of Figures

List of Definitions and Proofs

Chapter 1

Introduction

Transactions are the *de facto* synchronisation mechanism in modern distributed databases [1]. Distributed databases often implement weak transactional consistency guarantees known as *consistency models*, allowing them to achieve scalability and performance. Many consistency models were designed by engineers to meet application requirements in industry and specified using informal definitions specific to particular real-world reference implementations, e.g. [2, 3, 4, 5, 6, 7, 8, 9]. In recent years, general definitions of consistency models have been defined independently of particular implementations, either declaratively or operationally [10]. Xiong et al. [10] define a general operational semantics for weak consistency models. Our aim is to reformulate their semantics into a forward-looking specification that can be implemented efficiently.

Significant work has been done on declarative semantics for defining consistency models, with the most popular approaches being dependency graphs [11] and abstract executions [12, 13]. The operational approach for defining consistency models, unlike the declarative approach, has not been studied much [10]. Crooks et al. [14] proposed a state-based trace semantics over a global centralised store for describing weak consistency models [1]. Using their semantics, they demonstrate that multiple definitions of snapshot isolation (`SI`) are equivalent [1]. Nagar and Jagannathan [15] proposed a fine-grained interleaving operational semantics on abstract executions, building a model-checking tool for proving robustness results for client programs [1]. Kaki et al. [16] proposed an operational semantics for SQL transactions over an abstract, centralised, single-version store [10]. They develop a program logic and prototype tool for reasoning about client programs, but cannot capture models such as parallel snapshot isolation (`PSI`) and causal consistency (`CC`) which are important for distributed databases [1]. Xiong et al. [10] introduced an interleaving operational semantics directly updating abstract states. No work has been done on implementing operational semantics for distributed databases to automatically generate litmus tests. Hence, this was our initial goal. Recognising that the semantics of Xiong et al. [10] required a lot of work before this was possible, we focus on advancing the theory.

Xiong et al. [10] introduce an interleaving operational semantics for describing the client-observable behaviour of atomic transactions updating distributed key-value stores. Their semantics is formed around a notion of abstract states consisting of a *centralised key-value store* (kv-store) with multi-versioning and a *client view* [10]. A kv-store is *global* in that it records all versions of a key; and a client view is *partial* in that a client may see only a subset of the versions [10]. They use *execution tests* to capture the conditions determining whether a client can commit the next transaction. An execution step depends only on the abstract state, the read-write set of the atomic transaction, and an *execution test* [10]. These execution tests induce different consistency models. Rather than having to examine the whole execution trace, execution tests depends only on the current abstract state [10].

From a theoretical point of view, the work of Xiong et al. [10] successfully provides a compelling abstract interface between distributed databases and clients. Their semantics can be used to both verify reference implementations and analyse the behaviour of client programs with respect to a particular consistency model [10]. However, their use of execution tests on abstract states to commit transactional updates requires a lot of backwards analysis on past versions of the state. Despite their operational approach, this backwards analysis undermines the 'forwardness' advantage of more traditional operational semantics. Thus, from an implementation perspective, it is exceedingly complex. We aimed to improve on their approach.

We successfully give a forward-looking reformulation of the transactional update rule, identifying the notion of *minimal views*, and prove its equivalence to the original formulation. Still there is an inefficiency in the the shifting of the client view to incorporate environment steps when starting a transaction. We recognise this view shift as a key step to developing an elegant implementation.

We choose to focus on the causal consistency (`CC`) model rather than the full generality of the operational semantics of Xiong et al. [10]. The `CC` model states that, if a version written by transaction t is included in the view of a client before committing a transaction, then all versions that t observes must also be in the client view [10]. This model is unique in that it captures only the dependencies associated with the executing client, and does not depend on the specific transaction being executed. Causal consistency intuitively has a 'forwards feel', which is reflected in its execution test definition in [10]. In particular, given a client view in `CC` (that satisfies the `CC` execution test), then the minimal view resulting from an update of a transaction is also in `CC`. This property cannot hold for other consistency models, such as `PSI` and `SI`, which depend on the transaction being committed.

We clearly demonstrate how to directly incorporate the environment steps to obtain all the possible extended `CC` views of a client when starting a transaction in a forward way, given the minimal view after an update. We extend the idea of minimal views to snapshots, and illustrate an approach for using an altered definition of a snapshot to obtain all the snapshots of *meaningful views* (extended views that result in a newer snapshot) of a client executing a transaction. Last, we provide an algorithm for implementing our approach.

1.1 Project Creation and Evolution

As this is a research-based project, a key aspect throughout has been finding the interesting questions to answer. This process of creative thinking constantly improved at each stage of the project, and an intuition for identifying the questions to investigate was gradually developed.

The project started with the goal of building an implementation of the operational semantics of Xiong et al. [10] and using that implementation to develop ways for automatically generating litmus tests. As the depth of understanding of the semantics grew and the most important parts were recognised, it seemed more prudent to focus first on the reformulation of the semantics to exploit properties of operational semantics and specific consistency models. Thus, the project evolved from an implementation project to a theory-focused project.

1.2 Contributions

We provide a forward-looking reformulation of the semantics of an atomic transaction under an execution test in chapter 5, which we show is equivalent to the original. This is the result of our evaluation of the backwards analysis of transactional updates in the semantics of Xiong et al. [10].

We provide the foundations for an efficient future implementation in chapter 7 and recognise the most challenging components of the semantics from the point of view of an implementation.

We demonstrate a systematic way for extending client views to incorporate information from the environment under causal consistency in chapter 8, utilising the notion of minimal views produced by our reformulation.

Building upon our minimal views approach, we alter the definition of snapshots and present an algorithm for producing all kv-stores obtainable by executing a client program under causal consistency in section 8.3. Following our algorithm are suggestions for potential optimisations, both theoretical and implementation specific.

In chapter 9 we conclude our results and propose future extensions to this project.

1.3 Outline

Chapter 2 provides the technical background for understanding this project, giving an overview of some well-known consistency models, declarative and operational approaches, and previous work for litmus test generation.

Chapter 3 evaluates the semantics of Xiong et al. [10], focusing on the semantics of an atomic transaction and introducing the use of execution tests to describe consistency models.

In chapter 4 we look at the work of Xiong et al. [10] from an implementation perspective and identify areas to optimise. We suggest improvements to the semantics to eliminate the backwards analysis of transactional updates.

We use our new definitions to give a forward-looking reformulation of the semantics of an atomic transaction under an execution test in chapter 5, and show that it is equivalent to the original.

In chapter 6 we discuss the effect of using our new transactional update definition on consistency models with different view shift conditions.

Chapter 7 explains our original aim of developing an implementation and the first steps we took before deciding to move our focus to advancing the theory.

In chapter 8 we explore the causal consistency (`CC`) model. We focus on updating client views with environment information in a forward way and demonstrate how it can be done using our minimal views from chapter 5.

The conclusions of this project are summarised in chapter 9, including proposals for future work.

Chapter 2

Background

This chapter provides the technical background for understanding the project and its creation process. In section 2.1 we explain the role of consistency models and give examples of some well-known models. We discuss the main declarative formalisms in section 2.2. In section 2.3 we present litmus tests, giving examples of some well-known tests (section 2.3.1) and introducing previous work on automatic litmus test generation (section 2.3.2). We give a summary of previous work on operational semantics in section 2.4.

2.1 Consistency Models

A consistency model describes the set of all states reachable by executing client programs starting from a consistent state. It does so by defining the guarantees a distributed store provides on the ordering of read and write operations across the nodes of the system. In other words, a model describes the set of all valid histories of transactions.

Databases use transactions, allowing programmers to think about blocks of actions as executing in isolation [17]. A transaction groups multiple operations into an independent unit and ensures that consistency is maintained. In distributed systems, data is often replicated and partitioned over many geo-distributed nodes. Ensuring serialisability of distributed transactions is thus a very expensive task with a significant performance cost.

Many modern distributed databases implement weaker consistency models trading off consistency for performance. Reasoning about and designing such consistency models is difficult due to their complex and counter-intuitive behaviour [18].

The semantics of a model can be captured either by a reference implementation or described by its non-serialisable behaviours, called anomalies.

Serialisability (SER) This is the strongest and most intuitive consistency model. It requires that transactions execute in a total sequential order. That is, transactions appear to happen in an instant, one after another, without interleaving sub-operations from different transactions [1]. The first definitions of the serialisability model were given in terms of implementation strategies using locks [19, 20]. The idea of history was later introduced, defined as the trace of fine-grained read and write operations. Serialisability was then formally defined [19]: there exists an equivalent history where transactions are executed sequentially. This means that even when executed concurrently, the history would be equivalent to a history of the transactions if they were executed sequentially on a single machine (single-node database). Since distributed transactions may execute on different nodes/replicas, a transaction may read a stale value that has been updated by an earlier transaction whose changes have not propagated. For this reason, serialisability additionally requires that a transaction t would observe the effect of all transactions ordered before t [1].

SQL isolation levels Berenson et al. summarised the four SQL isolation levels in [4]: Read uncommitted, read committed, repeatable read and serialisability.

1. Read uncommitted allows transactions to read uncommitted values.
2. Read committed restricts transactions to only read committed values but allows reads in the same transaction of the same object to be different.
3. Repeatable read requires that a transaction read the same value for reads of the same object.

These SQL isolation levels were defined in ANSI SQL-92 based on phenomena each level excludes. This had the goal of providing implementation independent definitions, as opposed to specific lock-based implementations of the four isolation levels [21]. Berenson et al. [4] claimed that these are ambiguous. Adya [11] proposed using dependency graphs to specify consistency models and formalised the four definitions to disambiguate them. These isolation levels are still commonly used in centralised databases, even though they are tightly related to lock-based implementations [1]. In the distributed databases world, however, implementing these consistency models comes at a performance cost: read uncommitted, read committed and repeatable read all support the interleaving of sub-operations between transactions, requiring more complex synchronisation mechanisms between sites [1]. Serialisability requires consensus between all nodes, limiting the number of transactions that can be executed in parallel.

Snapshot Isolation (SI) Snapshot isolation is a common consistency model supported by many centralised databases alongside the four SQL isolation levels. Snapshot isolation takes a different approach to that of lock-based implementation, operating using a multi-versioning approach. Berenson et al. [4] provide an English description of snapshot isolation implementation in a centralised database using transaction snapshot time and commit time. A transaction takes a snapshot of the database at the snapshot time. It then operates on this snapshot, reading values from the snapshot rather than the database itself. At commit time, the transaction checks that no write conflicts have happened since the snapshot time and commits the final state of the snapshot to the database, discarding any intermediate values [1]. The multi-versioning approach, with each transaction working on its own snapshot, naturally works well in distributed databases due to their architecture. Snapshot isolation has been implemented in many distributed database systems including both replicated and partitioned databases [1].

Parallel Snapshot Isolation (`PSI`) Despite `SI` fitting well to the architecture of distributed databases, it still enforces that concurrent transactions view the updates from other transactions in the same order, requiring that transactions executing at different sites agree on the relative order of operations [1]. This coordination process could carry significant penalties on performance. Sovran et al. [9] proposed a weaker model, parallel snapshot isolation (`PSI`), to tackle this synchronisation problem. Their implementation strategy is based on ideas from the `SI` implementation. Transactions consist of three stages: snapshot, internal execution and commit. The difference from `SI` is that rather than taking a snapshot of the entire database, a transaction takes a snapshot of a single site. The commit is then done locally, with the changes propagating non-deterministically to the other replicas. Synchronisation delays may result in transactions executing at different sites observing updates in different orders. Saeida Ardekani et al. [2] formalised `PSI` using constraints on read and write operations of a transaction. They further provide a better multi-versioning replicated database implementation of `PSI` (NMSI), where each version holds a dependency set of metadata (dependence vector).

Many consistency models were devised to meet a specific application need and defined by a reference implementation. This makes it harder to reason about and compare consistency models, for example to determine whether one is more permissive than another or to prove equivalence of different implementations of the same model. The problem posed by specificity of implementation details lead researchers to formalise consistency model specifications using general semantics which are expressive enough to capture many consistency models in a unified way.

2.2 Declarative Semantics: Dependency Graphs and Abstract Executions

Dependency graphs and abstract executions are the two main general declarative formalisms.

Adya [11] proposed the first approach that applies techniques for specifying consistency conditions using graphs and different types of dependencies to defining ANSI and commercial isolation levels in an implementation-independent manner. Dependency graphs have nodes representing transactions which are connected by directed edges representing dependencies between the transactions. Earlier locking-based definitions and their equivalent graph-based definition are restricted in their generality.

In [12], abstract executions were introduced for specifying eventual and causal consistency. They used a visibility relation (VIS) and an arbitration relation (AR) to represent the transactions observable by another transaction and the order in which transactions take effect. If $(t, t') \in \mathsf{VIS}$ then the effects of t can be observed by t'. If $(t, t') \in \mathsf{AR}$ then the versions written by t' override the versions written by t. Abstract executions were adapted to atomic transactions in [13].

2.3 Litmus Tests

The initial goal of this project was implementing the operational semantics of Xiong et al. [10], and using it to construct litmus tests. The background research therefore included investigating existing methods to generate litmus tests. Although the focus of the project has shifted away from litmus test generation, we provide examples of litmus tests in section 2.3.1, and describe past approaches for automatic generation of litmus tests in section 2.3.2, to support future work.

2.3.1 Examples of Litmus Tests

Litmus tests are small parallel programs that illustrate subtle differences between consistency models by demonstrating behaviours observable under one consistency model and not another.

Example 2.1. *Consider the client program* $\mathtt{P_{LU}}$*:*

$$\mathtt{P_{LU}} \overset{def}{=} \quad cl_1 : [\mathtt{x} := [k]\,;\ [k] := \mathtt{x}{+}1] \ || \ cl_2 : [\mathtt{x} := [k]\,;\ [k] := \mathtt{x}{+}1]$$

The lookup operation $\mathtt{x} := [k]$ *reads the value of* k *in local variable* $\mathtt{x}$*, and the mutation operation* $[k] := \mathtt{x}{+}1$ *writes* $\mathtt{x}{+}1$ *to* k*. Transactions that execute* atomically *are wrapped in square brackets.*

We have two clients, cl_1 *and* cl_2*, each executing a single atomic transaction. Assume that each client executes its transaction on a different replica of a replicated database with a single key* k*. The initial value of key* k *is* 0 *in both replicas.*

Under serialisability ($\mathtt{SER}$*) we expect the final value of* k *to be* 2*, as each client increments it by* 1 *and the transactions execute atomically. However, under some weaker* *consistency models such as causal consistency (*$\mathtt{CC}$*), it is possible that both* cl_1 *and* cl_2 *read* 0 *as the value for* k *and update it to* 1 *at their replicas. The updates eventually propagate to other replicas, resulting in the final state having* 1 *as the value for* k *in both.*

This behaviour is known as the lost update *anomaly.*

Example 2.2. *Consider the client program* $\mathtt{P_{LF}}$*:*

$$\begin{aligned}\mathtt{P_{LF}} \overset{def}{=} cl_1 : \big[\mathtt{x} := [k_1]\,;[k_1] := \mathtt{x}+1\big]\,;\big[\mathtt{y} := [k_2]\,;[k_2] := \mathtt{y}+1\big]\\ \mid\mid cl_2 : \big[\mathtt{x} := [k_1]\,;\mathtt{y} := [k_2]\big] \mid\mid cl_3 : \big[\mathtt{x} := [k_1]\,;\mathtt{y} := [k_2]\big]\end{aligned}$$

Assume that we are using a replicated database with multi-versioning with only two keys, k_1 *and* k_2*. Suppose that client* cl_1 *executes its transactions first, updating both* k_1 *and* k_2 *with the value* 1*. Our database now has two versions for each key, the first with value* 0 *and the second* 1*. The second client* cl_2 *executes next, using a view that includes both versions of* k_1 *but only the first version of* k_2*. The values read by* cl_2 *are* 1 *for* k_1 *and* 0 *for* k_2*.* cl_3 *executes its transaction last, using a view that includes both versions of* k_2 *but only the first version of* k_1*. This means that* cl_3 *reads* 0 *for* k_1 *and* 1 *for* k_2*. That is, client* cl_2 *sees the increment of* k_1 *before that of* k_2*, while* cl_3 *sees the increment of* k_2 *before that of* k_1*.*

This is known as the long fork *anomaly.*

Due to the many subtle differences between consistency model specifications, it is difficult to manually identify these differences and construct appropriate litmus test to capture them.

2.3.2 Automatically Generating Litmus Tests

We discuss multiple approaches for automatic generation of litmus tests, all of which are based on declarative semantics. No work has been done on operational semantics implementations for automatic litmus test generation for distributed databases. Hence, this was the initial goal of this project.

Jade Alglave et al. [22] use a combination of hand-written tests and some additional sources such as [23] as a test base. They also generate new tests which are mostly variations of some existing tests. Their approach is based on systematic enumeration of sequences of intra-thread relations from one memory access to the next, including address dependencies, data dependencies, control dependencies and identity of addresses [24].

In [18], John Wickerson et al. demonstrate an approach to automatically compare memory consistency models (MCMs) using a constraint solver. They identify four questions involved in designing and understanding memory consistency models and show that these four questions correspond to instances of a general constraint satisfaction problem whose solution involves finding litmus tests (programs and a state). As the problem is intractable when phrased over programs, they rephrase the constraints to be over program executions, making them computationally feasible. While earlier tools typically produce all candidate executions given an input program and a memory consistency model [18], they use the executions found to construct litmus tests that have executions that capture interesting MCM properties as their candidate executions.

Sela Mador-Haim et al. [25] also use a SAT solver in their approach for generating litmus tests to distinguish memory consistency models. It works by systematically generating all possible multi-threaded programs up to a specified size bound given two specifications of MCMs, and then checking whether one model allows a behaviour that the other does not. If this is the case, they produce a minimal litmus test which demonstrates the difference. Unlike Wickerson's approach where a SAT solver is used to generate litmus tests, they only use it to check the behaviour of pre-generated tests [18].

2.4 Operational Semantics

Unlike declarative approaches, very little work has been done on operational semantics for defining weak consistency models for distributed atomic transactions. The works of Xiong et al. [10] and Xiong [1] identify [14, 15, 16] as the key papers.

Crooks et al. [14] proposed a state-based trace semantics over a global centralised store for describing weak consistency models [1]. Their semantics introduces concepts called read states and commit tests, which are similar to the client views and execution tests of Xiong et al. [10]. A single reduction step in their semantics depends on the entire history of the execution trace [10]. Using their semantics, they show the equivalence of multiple definitions of snapshot isolation [1].

Nagar and Jagannathan [15] proposed a fine-grained interleaving operational semantics on abstract executions [10]. They build a model-checking tool for proving robustness results for client programs [1]. To achieve this, they convert abstract executions to dependency graphs and check robustness violations on the dependency graphs [10].

Kaki et al. [16] proposed an operational semantics for SQL transactions over an abstract, centralised, single-version store [10]. Their semantics uses the consistency models given by the standard ANSI/SQL isolation levels [4]. Using their semantics, they develop a program logic and prototype tool for reasoning about client programs [10]. Since changes to the global store are immediately available to all clients under their framework, they cannot capture models such as `PSI` and `CC` that are important for distributed databases [1].

Implementation work of operational semantics for automatic litmus test generation has not been done. Hence, this was the original aim of this project. As we began to explore the interleaving operational semantics of Xiong et al. [10], we realised that a lot of work on the theory was required before an implementation was possible. Thus, we shifted our focus to improving the theory.

Chapter 3

Operational Model

Xiong et al. [10] suggest an interleaving operational semantics for describing the client-observable behaviour of atomic transactions on distributed key-value stores. The semantics is based on a notion of abstract state which comprises a *centralised key-value store* (kv-store) with *multi-versioning*, which is global in the sense that it records all the versions of a key, and *client views*, which are partial in the sense that clients see only a subset of the versions. An execution step depends simply on the abstract state, the read-write set of the atomic transaction, and an *execution test* which determines if a client with a given view is allowed to commit the transaction [10].

In section 3.1 we introduce the main definitions and ideas used in the semantics of Xiong et al. [10]. We explore the semantics of an atomic transaction under an execution test ET in section 3.2, and explain the use of execution tests to describe consistency models in section 3.3.

Notation We use $\mathtt{A} \ni a$ to denote that elements of A are ranged over by a and its variants $a', a_1, \cdots$.. The notation [A] denotes the set of lists of A, and $[a_0, \cdots, a_n]$ denotes a list. Given two lists $l, l' \in [\mathtt{A}]$, the notation $l :: l'$ denotes the concatenation of the two lists. Given an element $a \in \mathtt{A}$ and a list $l \in [\mathtt{A}]$, the notation $l :: a$ denotes appending the element to the end of the list. The notation $|l|$ denotes the size of the list. The notation $l\,[i \mapsto a]$ denotes the update of $(i+1)^{\text{th}}$ component to a. The notation $\mathtt{A} \to \mathtt{B}$, $\mathtt{A} \rightharpoonup \mathtt{B}$ and $\mathtt{A} \overset{fin}{\rightharpoonup} \mathtt{B}$ denotes the set of total, partial and partial finite functions from A to B respectively. For a function $f \in \mathtt{A} \to \mathtt{B}$, (similarly for $\mathtt{A} \rightharpoonup \mathtt{B}$ and $\mathtt{A} \overset{fin}{\rightharpoonup} \mathtt{B}$), $a \in \mathtt{A}$ and $b \in \mathtt{B}$, the notation $f\,[a \mapsto b]$ denotes the update of the function defined by:

$$f\,[a \mapsto b]\,(a') \overset{def}{=} \begin{cases} b & \text{if } a' = a, \\ f(a') & \text{otherwise.} \end{cases}$$

Given relations $\mathsf{r}, \mathsf{r}' \subseteq \mathtt{A} \times \mathtt{A}$, we write: $\mathsf{r}^?$, r^+ and r^* for its reflexive, transitive and reflexive-transitive closures of r, respectively; r^{-1} for its inverse; $a_1 \xrightarrow{\mathsf{r}} a_2$ for

$(a_1, a_2) \in \mathsf{r}$; and $\mathsf{r}; \mathsf{r}'$ for $\{(a_1, a_2) \mid \exists a. (a_1, a) \in \mathsf{r} \wedge (a, a_2) \in \mathsf{r}'\}$.

3.1 Abstract States: Key-Value Stores and Client Views

A kv-store comprises key-indexed lists of versions which record the history of the key with values and meta-data of the transactions that accessed it: the writer and readers [10].

Definition 3.1 (Client and transactional identifiers). *The set of* client identifiers, $\textsc{ClientID} \ni cl$, *is a countably infinite set. The set of* transaction identifiers, $\textsc{TxID} \ni t$, *is defined by* $\textsc{TxID} \overset{def}{=} \{t_0\} \uplus \{t_{cl}^n \mid cl \in \textsc{ClientID} \wedge n \geq 0\}$, *where* t_0 *denotes the* initialisation transaction *and* t_{cl}^n *identifies a transaction committed by client* cl *with* n *determining the client session order (definition 3.2).*

Subsets of $\textsc{TxID}$ are ranged over by $T, T', \cdots$. We let $\textsc{TxID}_0 \overset{def}{=} \textsc{TxID} \setminus \{t_0\}$.

Definition 3.2 (Session order). *The session order relation,* $\mathsf{SO} \subseteq \textsc{TxID} \times \textsc{TxID}$, *is defined by* $\mathsf{SO} \overset{def}{=} \{(t, t') \mid \exists cl, n, m. t = t_{cl}^n \wedge t' = t_{cl}^m \wedge n < m\}$.

Definition 3.3 (Kv-stores). *Assume a countably infinite set of* keys, $\textsc{Key} \ni k$, *and a countably infinite set of* values, $\textsc{Value} \ni v$, *which includes the keys and an* initialisation value v_0. *The set of* versions, $\textsc{Version} \ni \nu$, *is defined by* $\textsc{Version} \overset{def}{=} \textsc{Value} \times \textsc{TxID} \times \mathcal{P}(\textsc{TxID}_0)$. *A* kv-store *is a function* $\mathcal{K} : \textsc{Key} \rightarrow \mathsf{List}(\textsc{Version})$, *where* $\mathsf{List}(\textsc{Version}) \ni \mathcal{V}$ *is the set of lists of versions.*

As in [10], versions have the form $\nu{=}(v, t, T)$, where v is a value, the *writer* t identifies the transaction that wrote v, and the *reader set* T identifies the transactions that read v. We use $\mathsf{val}(\nu)$, $\mathsf{w}(\nu)$ and $\mathsf{rs}(\nu)$ to project the individual components of ν. Given a kv-store $\mathcal{K}$ and a transaction t, we write $t \in \mathcal{K}$ if t is either the writer or one of the readers of a version included in $\mathcal{K}$, $|\mathcal{K}(k)|$ for the length of the version list $\mathcal{K}(k)$, and $\mathcal{K}(k, i)$ for the i^{th} version of k.

Definition 3.4 (Well-formed kv-store). *A kv-store* $\mathcal{K}$ *is well-formed, written* $\texttt{WfKvs}(\mathcal{K})$, *if and only if*

$$\forall k. \mathsf{val}(\mathcal{K}(k, 0)) = v_0 \tag{3.1}$$

$$\forall k, i, j. (\mathsf{rs}(\mathcal{K}(k, i)) \cap \mathsf{rs}(\mathcal{K}(k, j)) \neq \emptyset \vee \mathsf{w}(\mathcal{K}(k, i)) = \mathsf{w}(\mathcal{K}(k, j))) \Rightarrow i = j \tag{3.2}$$

$$\forall k, i, j, t, t'. t = \mathsf{w}(\mathcal{K}(k, i)) \wedge t' \in \mathsf{rs}(\mathcal{K}(k, i)) \Rightarrow (t', t) \notin \mathsf{SO}^{?} \tag{3.3}$$

$$\forall k, i, j, t, t'. t = \mathsf{w}(\mathcal{K}(k, i)) \wedge t' = \mathsf{w}(\mathcal{K}(k, j)) \wedge i < j \Rightarrow (t', t) \notin \mathsf{SO}^{?} \tag{3.4}$$

where $R^?$ *denotes the reflexive closure of* R. *Let* $\mathsf{KVS} \ni \mathcal{K}$ *denotes the set of* well-formed kv-stores.

Equation (3.1) states that the version list for each key has an initialisation version carrying the initialisation value v_0, under the assumption that the initialisation version is written by the initialisation transaction t_0 and initially has an empty reader set. The second condition (eq. (3.2)) is called the *snapshot property* and ensures that a transaction reads and writes at most one version for each key. Equations (3.3) and (3.4) ensure that the kv-store agrees with the session order of clients. That is, a client cannot read a version of a key that has been written by a future transaction within the same session, and the order in which versions are written by a client must agree with its session order.

A global kv-store provides an abstract centralised description of updates associated with distributed kv-stores that is *complete* in that no update has been lost in the description [10]. By contrast, in both replicated and partitioned distributed databases, a client may have incomplete information about updates distributed between machines [1]. Xiong et al. [10] model this incomplete information by defining a *view* of the kv-store which provides a *partial* record of the updates observed by a client. They require that a client view be *atomic* in that it can see either all or none of the updates of a transaction.

Definition 3.5 (Views). *A* view *of a kv-store* $\mathcal{K} \in \mathsf{KVS}$ *is a function* $u \in \textsc{Views}(\mathcal{K}) \overset{def}{=} \textsc{Key} \to \mathcal{P}(\mathbb{N})$ *such that, for all* i, i', k, k'*:*

$$0 \in u(k) \land (i \in u(k) \Rightarrow 0 \leq i < |\mathcal{K}(k)|) \tag{3.5}$$
$$i \in u(k) \land \mathsf{w}(\mathcal{K}(k,i)) = \mathsf{w}(\mathcal{K}(k',i')) \Rightarrow i' \in u(k')\,. \tag{3.6}$$

Given two views $u, u' \in \textsc{Views}(\mathcal{K})$*, the order between them is defined by* $u \sqsubseteq u' \overset{\text{def}}{\Leftrightarrow} \forall k \in \operatorname{dom}(\mathcal{K}).\, u(k) \subseteq u'(k)$*. The set of views is* $\textsc{Views} \overset{def}{=} \bigcup_{\mathcal{K}\in\mathsf{KVS}} \textsc{Views}(\mathcal{K})$*. The* initial view*,* u_0*, is defined by* $u_0(k) = \{0\}$ *for every* $k \in \textsc{Key}$*.*

A view is well-formed, written `WfView` $(\mathcal{K}, u)$, if it contains the initial version of each key (eq. (3.5)), the indices are in range (eq. (3.5)), and it is *atomic*, meaning that a client can observe either all or none of the updates of a transaction (eq. (3.6)).

The operational semantics updates pairs comprising a kv-store and a function describing the views of a finite set of clients, called *configurations*.

Definition 3.6 (Configurations). *A* configuration*,* $\Gamma \in \textsc{Conf}$*, is a pair* $(\mathcal{K},\mathcal{U})$ *with* $\mathcal{K} \in \mathsf{KVS}$ *and* $\mathcal{U} : \textsc{ClientID} \overset{fin}{\rightharpoonup} \textsc{Views}(\mathcal{K})$*. The set of* initial configurations*,* $\textsc{Conf}_0 \subseteq \textsc{Conf}$*, contains configurations of the form* $(\mathcal{K}_0,\mathcal{U}_0)$*, where* $\mathcal{K}_0$ *is the initial kv-store defined by* $\mathcal{K}_0(k) \overset{def}{=} (v_0, t_0, \emptyset)$ *for all* $k \in \textsc{Key}$*.*

Given a configuration $(\mathcal{K},\mathcal{U})$ and a client cl, if $u = \mathcal{U}(cl)$ is defined then, for each k, the configuration determines the sub-list of versions in $\mathcal{K}$ that cl sees. If indices

$$\textsc{CAtomicTrans}\quad \frac{\begin{array}{c} u \sqsubseteq u'' \quad \sigma = \mathsf{snapshot}(\mathcal{K}, u'') \quad (s, \sigma, \emptyset), \mathtt{T} \rightsquigarrow^* (s', \_, \mathcal{F}), \mathtt{skip} \quad \mathtt{canCommit}_{\mathtt{ET}}(\mathcal{K}, u'', \mathcal{F}) \\ t \in \mathsf{NextTxID}(cl, \mathcal{K}) \qquad \mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, u'', \mathcal{F}, t) \qquad \mathtt{vShift}_{\mathtt{ET}}(\mathcal{K}, u'', \mathcal{K}', u') \end{array}}{cl \vdash (\mathcal{K}, u, s), [\mathtt{T}] \xrightarrow{(cl, u'', \mathcal{F})}_{\mathtt{ET}} (\mathcal{K}', u', s'), \mathtt{skip}}$$

Figure 3.1: The semantics of an atomic transaction under execution test ET [10]

$i, j \in u(k)$ and $i < j$, then client cl sees the values carried by versions $\mathcal{K}(k, i)$ and $\mathcal{K}(k, j)$, and it also sees that the version $\mathcal{K}(k, j)$ is more up-to-date than $\mathcal{K}(k, i)$. This allows to associate a *snapshot* with the view u, which identifies, for each key k, the last version included in the view. This definition assumes that the database satisfies the *last-write-wins* resolution policy, employed by many distributed key-value stores.

Definition 3.7 (View snapshots). *Given $\mathcal{K} \in$ KVS and $u \in \textsc{Views}(\mathcal{K})$, the* snapshot *of u in $\mathcal{K}$ is a function,* $\mathsf{snapshot}(\mathcal{K}, u) : \textsc{Key} \rightarrow \textsc{Value}$*, defined by:*

$$\mathsf{snapshot}(\mathcal{K}, u) \overset{def}{=} \lambda k.\, \mathsf{val}(\mathcal{K}(k, \max_<(u(k))))$$

where $\max_<(u(k))$ is the maximum element in $u(k)$ w.r.t. the natural order $<$ over $\mathbb{N}$.

The effects of executing a transaction T on a snapshot σ of kv-store $\mathcal{K}$ are captured in a *fingerprint*. A fingerprint of T on snapshot σ captures the first values T reads from σ, and the last values T writes to σ and intends to commit to $\mathcal{K}$.

Definition 3.8 (Fingerprints). *Let* $\textsc{Op}$ *denote the set of* read (R) and write (W) operations *defined by* $\textsc{Op} \overset{def}{=} \{(l, k, v) \mid l \in \{\mathtt{R}, \mathtt{W}\} \wedge k \in \textsc{Key} \wedge v \in \textsc{Value}\}$*. A* fingerprint $\mathcal{F}$ *is a set of operations, $\mathcal{F} \subseteq \textsc{Op}$, such that: $\forall k \in \textsc{Key}, l \in \{\mathtt{R}, \mathtt{W}\}.\, (l, k, v_1), (l, k, v_2) \in \mathcal{F} \Rightarrow v_1 = v_2$.*

A fingerprint can have at most one read operation and at most one write operation for a given key. This follows the snapshot property assumption: reads are taken from a single snapshot of the kv-store; and only the last write of a transaction to each key is committed to the kv-store.

We focus on the semantics of an atomic transaction, described by the CATOMICTRANS rule. All rules for command and program semantics are provided in appendix A.

3.2 The CATOMICTRANS Rule

The CATOMICTRANS describes the steps of execution of an atomic transaction under the execution test ET. We explain the premises of the rule and identify areas that can be improved to make an implementation of the rule more computationally forward.

ET	$\texttt{canCommit}_{\texttt{ET}}(\mathcal{K}, u, \mathcal{F}) \overset{def}{=} \texttt{closed}(\mathcal{K}, u, R_{\texttt{ET}})$	$\texttt{vShift}_{\texttt{ET}}(\mathcal{K}, u, \mathcal{K}', u')$
MR	true	$u \sqsubseteq u'$
RYW	true	$\forall t \in \mathcal{K}' \setminus \mathcal{K}. \forall k, i. (\mathsf{w}(\mathcal{K}'(k,i)), t) \in \mathsf{SO}^{?} \Rightarrow i \in u'(k)$
CC	$R_{\texttt{CC}} \overset{def}{=} \mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}$	$\texttt{vShift}_{\texttt{MR} \cap \texttt{RYW}}(\mathcal{K}, u, \mathcal{K}', u')$
UA	$R_{\texttt{UA}} \overset{def}{=} \bigcup_{(\texttt{W},k,\_) \in \mathcal{F}} \mathsf{WW}_{\mathcal{K}}^{-1}(k)$	true
PSI	$R_{\texttt{PSI}} \overset{def}{=} R_{\texttt{UA}} \cup R_{\texttt{CC}} \cup \mathsf{WW}_{\mathcal{K}}$	$\texttt{vShift}_{\texttt{MR} \cap \texttt{RYW}}(\mathcal{K}, u, \mathcal{K}', u')$
CP	$R_{\texttt{CP}} \overset{def}{=} \mathsf{SO}; \mathsf{RW}_{\mathcal{K}}{}^{?} \cup \mathsf{WR}_{\mathcal{K}}; \mathsf{RW}_{\mathcal{K}}{}^{?} \cup \mathsf{WW}_{\mathcal{K}}$	$\texttt{vShift}_{\texttt{MR} \cap \texttt{RYW}}(\mathcal{K}, u, \mathcal{K}', u')$
SI	$R_{\texttt{SI}} \overset{def}{=} R_{\texttt{UA}} \cup R_{\texttt{CP}} \cup (\mathsf{WW}_{\mathcal{K}}; \mathsf{RW}_{\mathcal{K}})$	$\texttt{vShift}_{\texttt{MR} \cap \texttt{RYW}}(\mathcal{K}, u, \mathcal{K}', u')$
SER	$R_{\texttt{SER}} \overset{def}{=} \mathsf{WW}_{\mathcal{K}}^{-1}$	true

Figure 3.2: Execution tests of consistency models defined by `canCommit` and `vShift` predicates [10], where SO is as given in definition 3.2

Advancing the view The first premise allows the executing client to advance its view u to a newer view u'' to incorporate environment information. For example, suppose that a client cl has the initial view u_0 over a kv-store $\mathcal{K}$ and is about to execute a transaction incrementing the value of key k. It might be that another client cl' has committed a newer version to k. Advancing the view allows client cl to have a more up-to-date view of the kv-store. To advance our view to u'' it must be allowed under the execution test ET as determined by the `canCommit` predicate (section 3.2).

Snapshot and fingerprint Using the advanced view u'', a snapshot of the kv-store $\mathcal{K}$ is taken. This snapshot is used to accumulate the fingerprint $\mathcal{F}$ and update the stack as the transaction executes locally to completion. Once the effect ot the transaction is recorded in the fingerprint, the snapshot is no longer needed.

The rules for extending the fingerprint are described by the *combination operator* $\lll : \mathcal{P}(\textsc{Op}) \times (\textsc{Op} \uplus \{\epsilon\}) \to \mathcal{P}(\textsc{Op})$ (fig. A.1). This is part of the TPRIMITIVE rule on primitive transactions. A read operation from k is added if $\mathcal{F}$ contains no entry for k, and a write operation to k is always added to $\mathcal{F}$, removing previous writes to k. This ensures that the snapshot property (eq. (3.2)) is satisfied.

Can-commit check The $\texttt{canCommit}_{\texttt{ET}}(\mathcal{K}, u'', \mathcal{F})$ premise checks whether the fingerprint $\mathcal{F}$ of the transaction is compatible with the kv-store $\mathcal{K}$ and the client view u'' under the execution test ET. If it is, then the transaction *can commit*. `canCommit` is parametric in the execution test ET, meaning that the conditions checked upon committing depend on the consistency model under which the transaction is to commit. Definitions of `canCommit` for several execution tests associated with well-known consistency models are given in fig. 3.2.

The $\texttt{canCommit}_{\texttt{ET}}$ predicate requires that the view of the executing client is closed with respect to a relation R on transactions in the kv-store $\mathcal{K}$. This is defined by the *prefix-closure* on the *set of visible transactions*.

Definition 3.9 (Visible transactions)**.** *The* set of visible transactions *of a kv-store $\mathcal{K}$ and a view u is defined by:*

$$\mathsf{visTx}\left(\mathcal{K}, u\right) \overset{def}{=} \left\{\mathsf{w}\left(\mathcal{K}\left(k, i\right)\right) \mid i \in u\left(k\right)\right\}.$$

That is, $\mathsf{visTx}\left(\mathcal{K}, u\right)$ is the set of all writers of versions that are included in the view u.

Definition 3.10 (Prefix closure)**.** *Given a binary relation on transactions, $R \subseteq \text{TXID} \times \text{TXID}$, a view u is* closed *with respect to a kv-store $\mathcal{K}$ and R, written* `closed` $\left(\mathcal{K}, u, R\right)$*, if and only if:*

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

That is, if transaction t is visible in u, ($t \in \mathsf{visTx}\left(\mathcal{K}, u\right)$), then all writing transactions t', (not read-only $t' \notin \left\{t'' \mid \forall k, i.\, t'' \neq \mathsf{w}\left(\mathcal{K}\left(k, i\right)\right)\right\}$), that are R^*-before t, ($t' \in \left(R^*\right)^{-1}\left(t\right)$), are also visible in u, ($t' \in \mathsf{visTx}\left(\mathcal{K}, u\right)$).

Computing the prefix-closure every time a transaction is executed is very computationally expensive. Moreover, any candidate view from the first premise (section 3.2) needs to be checked to determine its compatibility before the transaction can commit. In chapter 8 we investigate different approaches to cleverly advance a view in a way that guarantees `canCommit` is satisfied under causal consistency.

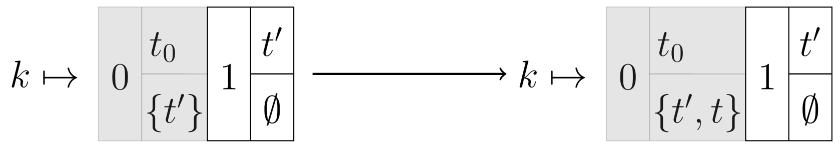
Next transaction identifier The client cl with view u'' over the kv-store $\mathcal{K}$ is now ready to commit the transaction with fingerprint $\mathcal{F}$ to the kv-store. The next premise makes use of the set $\mathsf{NextTxID}\left(cl, \mathcal{K}\right)$ to pick the next transaction identifier.

Definition 3.11 (Next transaction identifiers)**.** *Given a kv-store $\mathcal{K} \in \text{KVS}$, the set of* next available transaction identifiers *for a client cl, written $\mathsf{NextTxID}\left(cl, \mathcal{K}\right)$, is defined by:*

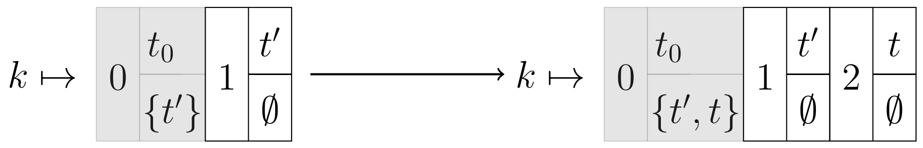
$$\mathsf{NextTxID}\left(cl, \mathcal{K}\right) \overset{def}{=} \left\{t_{cl}^{n} \mid t_{cl}^{n} \in \text{TXID} \land \forall m \in \mathbb{N}.\, \forall t_{cl}^{m} \in \text{TXID}.\, t_{cl}^{m} \in \mathcal{K} \Rightarrow m < n\right\}.$$

That is, pick any transaction identifier that is greater than all previous transactions committed by client cl. This ensures that session order SO can be determined by the transaction identifiers.

Notice that allowing any transaction identifier greater than previously committed transactions of the executing client to be picked as the next transaction identifier introduces unnecessary non-determinism. In section 4.1 we define the $\mathsf{FreshTxid}$ function to deterministically select the next consecutive available transaction identifier for a client.



(a) An example of UpdateKV $(\mathcal{K}, u, \{(\mathtt{R}, k, 0)\} \uplus \mathcal{F}, t)$ (view u is highlighted)



(b) An example of UpdateKV $(\mathcal{K}, u, \{(\mathtt{W}, k, 2)\} \uplus \mathcal{F}, t)$ (view u is highlighted)

Figure 3.3: An example of UpdateKV [1]

Transactional update The function UpdateKV $(\mathcal{K}, u, \mathcal{F}, t)$ describes the effect of executing transaction t with fingerprint $\mathcal{F}$ under view u on kv-store $\mathcal{K}$. 1. for each read $(\mathtt{R}, k, v) \in \mathcal{F}$, it adds t to the reader set of the last version of k in u (fig. 3.3a); 2. for each write $(\mathtt{W}, k, v) \in \mathcal{F}$, it appends a new version $(v, t, \emptyset)$ to $\mathcal{K}(k)$ (fig. 3.3b).

Definition 3.12 (Transactional update). *The function* UpdateKV $(\mathcal{K}, u, \mathcal{F}, t)$ *is defined by:*

$$\mathsf{UpdateKV}(\mathcal{K}, u, \emptyset, t) \overset{def}{=} \mathcal{K}$$

$$\mathsf{UpdateKV}(\mathcal{K}, u, \{(\mathtt{R}, k, v)\} \uplus \mathcal{F}, t) \overset{def}{=} \textit{let } i = \max_{<}(u(k)) \textit{ and } (v, t', T) = \mathcal{K}(k, i) \textit{ in } \mathsf{UpdateKV}(\mathcal{K}[k \mapsto \mathcal{K}(k)[i \mapsto (v, t', T \uplus \{t\})]], u, \mathcal{F}, t)$$

$$\mathsf{UpdateKV}(\mathcal{K}, u, \{(\mathtt{W}, k, v)\} \uplus \mathcal{F}, t) \overset{def}{=} \textit{let } \mathcal{K}' = \mathcal{K}[k \mapsto \mathcal{K}(k) :: (v, t, \emptyset)] \textit{ in } \mathsf{UpdateKV}(\mathcal{K}', u, \mathcal{F}, t)$$

where, given a list of versions $\mathcal{V} = \nu_0 :: \cdots :: \nu_n$ *and an index* $i : 0 \leq i \leq n$, *then* $\mathcal{V}[i \mapsto \nu] \overset{def}{=} \nu_0 :: \cdots :: \nu_{i-1} :: \nu :: \nu_{i+1} :: \cdots :: \nu_n$.

At each step we choose an operation from the fingerprint, in any order, apply it to the kv-store to reflect its effect and recursively call UpdateKV with the updated kv-store and the rest of the fingerprint, with the operation removed.

View shift The `vShift` predicate is used to check whether a *view shift* is permitted under execution test `ET`. It is used to restrict the permitted views of a client over the resulting kv-store after an application of a transaction. Given an initial kv-store $\mathcal{K}$ and a view $u \in \textsc{Views}(\mathcal{K})$, and the resulting kv-store $\mathcal{K}'$ and view $u' \in \textsc{Views}(\mathcal{K}')$ after a transactional update, $\mathtt{vShift}_{\mathtt{ET}}(\mathcal{K}, u, \mathcal{K}', u')$ determines if shifting the client view from u to u' is allowed under `ET`. Notice that `vShift` is parametric in the execution test `ET`, meaning that the conditions checked depend on the consistency model, similarly to `canCommit`. Definitions of `vShift` for several execution tests associated with well-known consistency models are given in fig. 3.2.

Note that the `vShift` check does not define how to obtain a new view that satisfies the view-shift conditions (section 3.2). To find such a view, we would need to generate candidate views and discard those for which the conditions do not hold.

Many consistency models define `vShift` as the conjunction of the `MR` and `RYW` session guarantees (fig. 3.2). We are therefore particularly interested in looking at $\texttt{vShift}_{\texttt{MR}}$ and $\texttt{vShift}_{\texttt{RYW}}$.

View shift - Read Your Writes (`RYW`) This consistency model is explained in [10] as follows:

> This consistency model states that a client must always see all the versions written by the client itself. The $\texttt{vShift}_{\texttt{RYW}}$ predicate thus states that after executing a transaction, a client contains all the versions it wrote in its view. This ensures that such versions will be included in the view of the client when committing future transactions.

The predicate is defined on the difference of kv-stores before and after executing a transaction using SO:

$$\texttt{vShift}_{\texttt{RYW}}(\mathcal{K}, u, \mathcal{K}', u') \overset{def}{=} \forall t \in \mathcal{K}' \setminus \mathcal{K}.\, \forall k, i.\, (\mathsf{w}(\mathcal{K}'(k,i)), t) \in \mathsf{SO}^{?} \Rightarrow i \in u'(k).$$

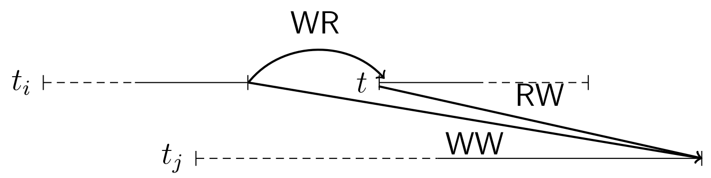
That is, given an initial kv-store $\mathcal{K}$ and view $u \in \textsc{Views}(\mathcal{K})$, and the resulting kv-store after a transactional update $\mathcal{K}'$, the view shift from the initial view to $u' \in \textsc{Views}(\mathcal{K}')$ is allowed under `RYW`. Notice that because $\mathcal{K}'$ is the result of a transactional update, the set difference $\mathcal{K}' \setminus \mathcal{K}$ contains only a single element - the committed transaction.

With this definition, we are restricted to reasoning about `RYW` associated with a single transactional update. We define a new predicate $\texttt{RYW}(cl, \mathcal{K}, u)$ (definition 4.3) in section 4.2 to decouple the `RYW` guarantees from a specific transactional update.

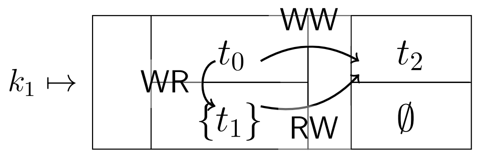
View Shift - Monotonic Read (`MR`) The Monotonic Read consistency model states that a client cannot lose information when committing. That is, a client can only observe more up-to-date versions from a kv-store, expanding its view. The `vShift` predicate for `MR` is therefore defined as $\texttt{vShift}_{\texttt{MR}}(\mathcal{K}, u, \mathcal{K}', u') \overset{def}{=} u \sqsubseteq u'$.

3.3 Execution Tests and Consistency Models

Xiong et al. use the notion of an *execution test*, specifying whether a client is allowed to commit a transaction in a given kv-store.



(a) An example of dependencies between transactions with respect to the time line of the starts and commits of these transactions (dashed line can be stretched)



(b) Dependencies of transactions in kv-stores (values omitted)

Figure 3.4: Dependency relations on key-value stores [10]

Definition 3.13 (Execution tests). *An* execution test, ET*, is a set of tuples,* $\mathtt{ET} \subseteq \mathrm{KVS} \times \textsc{Views} \times \textsc{Fp} \times \mathrm{KVS} \times \textsc{Views}$*, such that for all* $(\mathcal{K}, u, \mathcal{F}, \mathcal{K}', u') \in \mathtt{ET}$*: 1.* $u \in \textsc{Views}(\mathcal{K})$ *and* $u' \in \textsc{Views}(\mathcal{K}')$*; 2.* $\mathtt{canCommit}_{\mathtt{ET}}(\mathcal{K}, u, \mathcal{F})$*; 3.* $\mathtt{vShift}_{\mathtt{ET}}(\mathcal{K}, u, \mathcal{K}', u')$*; and 4. for all* $k \in \mathcal{K}$ *and* $v \in \textsc{Value}$*, if* $(\mathtt{R}, k, v) \in \mathcal{F}$ *then* $\mathcal{K}(k, \max_<(u(k))) = v$*.*

Intuitively, $(\mathcal{K}, u, \mathcal{F}, \mathcal{K}', u') \in \mathtt{ET}$ captures that, under the execution test ET, a client with initial view u over kv-store $\mathcal{K}$ can commit a transaction with fingerprint $\mathcal{F}$ to obtain the resulting kv-store $\mathcal{K}'$ while shifting its view to u' [10]. The last condition in definition 3.13 enforces the last-write-wins policy: a transaction always reads the most recent writes from the initial view u.

An execution test ET induces a consistency model as the set of kv-stores obtained by having clients non-deterministically commit transactions, so long as the constraints imposed by ET are satisfied [10]. Definitions of `canCommit` and `vShift` for multiple consistency models are given in fig. 3.2.

Definition 3.14 (Consistency models). *The* consistency model *induced by an execution test* ET *is defined as:* $\mathsf{CM}(\mathtt{ET}) \overset{def}{=} \left\{\mathcal{K} \mid \exists \mathcal{K}_0, \mathcal{U}_0, \mathcal{E}, \mathtt{P}. (\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}), \mathtt{P} \rightarrow^*_{\mathtt{ET}} (\mathcal{K}, \_, \_), \_\right\}$.

We provide definitions for the dependency relations *write-read* (WR), *write-write* (WW), and *read-write* (RW). These, together with the session order (SO) relation (definition 3.2) are used to define the `canCommit` and `vShift` conditions of consistency models (fig. 3.2).

Figure 3.4b shows dependency relation between transactions on an example kv-store.

Definition 3.15 (Dependency relations on kv-stores). *Given a kv-store* $\mathcal{K}$*, a key* k *and indexes* i, j *such that* $0 \leq i < j < |\mathcal{K}(k)|$*, if there exists* t_i, T_i, t *such that* $\mathcal{K}(k, i) = (\_, t_i, T_i)$*,* $\mathcal{K}(k, j) = (\_, t_j, \_)$ *and* $t \in T_i$*, then for every key* k*:*

1. *there is a* Write-Read *dependency from* t_i *to* t*, written* $(t_i, t) \in \mathsf{WR}_{\mathcal{K}}(k)$*;*
2. *there is a* Write-Write *dependency from* t_i *to* t_j*, written* $(t_i, t_j) \in \mathsf{WW}_{\mathcal{K}}(k)$*; and*
3. *if* $t \neq t_j$*, then there is a* Read-Write *anti-dependency from* t *to* t_j*, written* $(t, t_j) \in \mathsf{RW}_{\mathcal{K}}(k)$.

Item 1 intuitively means that t_i commits before t starts, as depicted in fig. 3.4a; Item 2 intuitively means that t_i commits before t_j commits, as depicted in fig. 3.4a; and item 3 intuitively means that t starts before t_j commits, as depicted in fig. 3.4a.

Chapter 4

Identifying the Gaps

We identify areas to optimise in the semantics and suggest improvements and modifications to allow for a smarter and more efficient implementation.

In section 4.1 we suggest a deterministic alternative to the NextTxID function for selecting the next transaction identifier for a client. We explain the 'backwardness' of the `vShift` predicate in section 4.2, and update our implementation of a transactional update to address this backwardness in section 4.3. We use our new definitions to reformulate the CATOMICTRANS rule in section section 5.1, and prove its equivalence to the original rule.

4.1 NextTxID

We notice that the definition of NextTxID allows for a non-deterministic choice of a new transaction identifier as any n greater than the existing transaction identifiers of the client can be selected. As we aim to advance towards a more implementable semantics we would like to, as much as possible, take a 'forward' approach to reasoning about executions. Therefore, we choose to define instead a function $\mathsf{FreshTxid}\,(cl, \mathcal{K})$ to return a fresh transaction identifier, rather than a set, for client cl. We intend to use this new definition to make the implementation of selecting a new transaction identifier deterministic.

Definition 4.1 (Fresh transaction identifier). *Given a kv-store $\mathcal{K} \in$ KVS and a client identifier cl, the function* $\mathsf{FreshTxid}\,(cl, \mathcal{K})$ *returning the* next available transaction identifier *for client cl is defined by:*

$$\mathsf{FreshTxid}\,(cl, \mathcal{K}) \overset{def}{=} \begin{cases} t_{cl}^{0}, & \textit{if } \forall n.\, t_{cl}^{n} \notin \mathcal{K} \\ t_{cl}^{n+1}, & \textit{if } t_{cl}^{n} \in \mathcal{K} \wedge \forall m.\, t_{cl}^{m} \in \mathcal{K} \Rightarrow m \leq n \end{cases}$$

The first case deals with the first transaction committed by client cl, where no previous transactions are in $\mathcal{K}$. The second case handles any following transaction by incrementing the largest (latest) transaction executed by cl by one.

By selecting the next consecutive identifier for a new client transaction we eliminte the non-determinism and simplify our reasoning, for example when looking at closure properties as knowing the maximum transaction for a client is enough to infer all of its previous transaction.

We show that our new definition is allowed in the original rule by demonstrating that the transaction identifier returned by $\mathsf{FreshTxid}\,(cl, \mathcal{K})$ is a member of the set $\mathsf{NextTxID}\,(cl, \mathcal{K})$. In particular, using $\mathsf{FreshTxid}\,(cl, \mathcal{K})$ is equivalent to selecting the minimum transaction identifier in the original $\mathsf{NextTxID}\,(cl, \mathcal{K})$ definition.

Proposition 4.2 (FreshTxid is in NextTxID). *Given a client identifier cl and a kv-store $\mathcal{K}$, the transaction identifier returned by* $\mathsf{FreshTxid}\,(cl, \mathcal{K})$ *is a member of the set* $\mathsf{NextTxID}\,(cl, \mathcal{K})$*,* $\mathsf{FreshTxid}\,(cl, \mathcal{K}) \in \mathsf{NextTxID}\,(cl, \mathcal{K})$.

Proof. We have two cases to consider:

Case $\forall n.\, t_{cl}^n \notin \mathcal{K}$. The antecedent in definition 3.11 is false in this case, making the set returned by NextTxID equal to $S = \{t_{cl}^n \mid n \in \mathbb{N}\}$. FreshTxid returns t_{cl}^0, the minimum transaction identifier in S.

Case $\exists n.\, t_{cl}^n \in \mathcal{K}$. Let t_{cl}^m be the latest transaction executed by client cl on $\mathcal{K}$. The set returned by NextTxID contains all transaction identifiers t_{cl}^n with $n \geq m+1$ by definition. FreshTxid returns t_{cl}^{m+1}, the minimum transaction identifier in NextTxID.

We have shown that

$$\mathsf{FreshTxid}\,(cl, \mathcal{K}) = \min_{<}(\mathsf{NextTxID}\,(cl, \mathcal{K}))$$

where $\min_{<}(\mathsf{NextTxID}\,(cl, \mathcal{K}))$ is the minimum element in $\mathsf{NextTxID}\,(cl, \mathcal{K})$ w.r.t. the natural order $<$ over $\mathbb{N}$. □

4.2 vShift

Observe that the `vShift` check does not define how to obtain a new view that satisfies the view-shift conditions (section 3.2). This means that to find such a view, we would need to generate candidate views and eliminate those for which the conditions do not hold. Moreover, since the check is parametric in the execution test `ET`, the conditions differ between different consistency models. However, Xiong et al. [10] define `vShift` for `CC`, `CP`, `PSI` and `SI` as the conjunction of the `MR` and `RYW` session guarantees (fig. 3.2).

Using the current definition of $\texttt{vShift}_{\texttt{RYW}}$ we can only talk about RYW properties with respect to an update of a single transaction. However, we would like to be able to determine whether RYW session guarantees are satisfied given any client cl with view u over a kv-store $\mathcal{K}$.

We define a new predicate $\texttt{RYW}(cl, \mathcal{K}, u)$ to check whether RYW session guarantees are met given a client cl with view u over kv-store $\mathcal{K}$.

Definition 4.3 (Client Read Your Writes). *Given a client identifier cl, a kv-store $\mathcal{K}$, and a view over the store $u \in \textsc{Views}(\mathcal{K})$, the client view is valid under* RYW *if it satisfies*

$$\texttt{RYW}(cl, \mathcal{K}, u) \overset{def}{=} \forall n \in \mathbb{N}. \forall t_{cl}^{n} \in \textsc{TxID}_0. \left(\forall k, i. \, \mathsf{w}(\mathcal{K}(k, i)) = t_{cl}^{n} \Rightarrow i \in u(k) \right).$$

Intuitively, given a kv-store $\mathcal{K}$ and a view u for client cl, all versions written by client cl's transactions are observable in its view.

Since clients can advance their view at the beginning of a new transaction it is enough to shift the view of a client after a transactional update to the *minimal view* that satisfies the vShift check as we show in section 5.1 (definition 5.1).

4.3 Transactional Update and View Shift

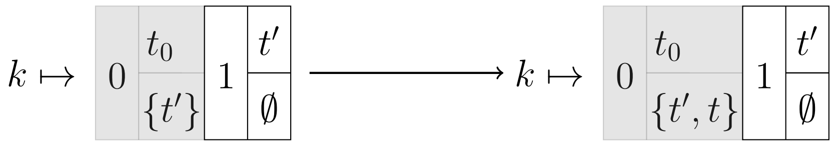
The CAtomicTrans in fig. 3.1 uses the vShift predicate to define what view-shifts are allowed under an execution test ET. Currently, finding a view that satisfies the vShift predicate requires generating some view u' and computing whether the shift from the initial view to u' satisfies the view-shift conditions. In our implementation we prefer to avoid backward computations and instead aim to take advantage of the operational semantics properties to calculate ahead.

The vShift conditions for CC, CP, PSI and SI are all defined by $\texttt{vShift}_{\texttt{MR}\cap\texttt{RYW}}$, the conjunction of the $\texttt{vShift}_{\texttt{MR}}$ and $\texttt{vShift}_{\texttt{RYW}}$ conditions. We suggest a new implementation of UpdateKV incorporating the $\texttt{vShift}_{\texttt{MR}\cap\texttt{RYW}}(\mathcal{K}, u, \mathcal{K}', u')$ immediately (definition 4.4). Using our new definition, we redefine the CAtomicTrans rule (fig. 5.1) and prove its equivalence to the original rule for these consistency models.

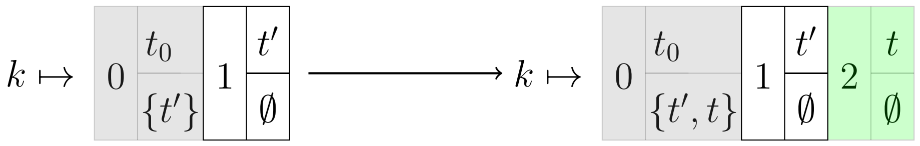
Definition 4.4 (New transactional update). *Given a kv-store $\mathcal{K} \in$ KVS, views $u_r, u_w \in \textsc{Views}(\mathcal{K})$, a fingerprint $\mathcal{F}$, and a transaction identifier $t \in \textsc{TxID}_0$, the transactional update function* $\mathsf{NewUpdateKV}(\mathcal{K}, u_r, u_w, \mathcal{F}, t)$ *is defined by:*

$$\mathsf{NewUpdateKV}(\mathcal{K}, u_r, u_w, \emptyset, t) \overset{def}{=} (\mathcal{K}, u_w)$$

$$\mathsf{NewUpdateKV}(\mathcal{K}, u_r, u_w, \{(\mathtt{R}, k, v)\} \uplus \mathcal{F}, t) \overset{def}{=}$$



(a) An example of NewUpdateKV $(\mathcal{K}, u_r, u_w, \{(\mathtt{R}, k, 0)\} \uplus \mathcal{F}, t)$



(b) An example of NewUpdateKV $(\mathcal{K}, u_r, u_w, \{(\mathtt{W}, k, 2)\} \uplus \mathcal{F}, t)$

Figure 4.1: An example of NewUpdateKV (u_r in gray, u_w in green)

$$\texttt{let } i = \max_<(u_r(k)) \texttt{ and } (v, t', T) = \mathcal{K}(k, i) \texttt{ in}$$
$$\mathsf{NewUpdateKV}(\mathcal{K}[k \mapsto \mathcal{K}(k)[i \mapsto (v, t', T \uplus \{t\})]], u_r, u_w, \mathcal{F}, t)$$

$$\mathsf{NewUpdateKV}(\mathcal{K}, u_r, u_w, \{(\mathtt{W}, k, v)\} \uplus \mathcal{F}, t) \overset{def}{=}$$
$$\texttt{let } \mathcal{K}' = \mathcal{K}[k \mapsto \mathcal{K}(k) :: (v, t, \emptyset)] \texttt{ and } u'_w = u_w[k \mapsto u_w(k) \uplus \{|\mathcal{K}(k)|\}] \texttt{ in}$$
$$\mathsf{NewUpdateKV}(\mathcal{K}', u_r, u'_w, \mathcal{F}, t)$$

where, given a list of versions $\mathcal{V} = \nu_0 :: \cdots :: \nu_n$ *and an index* $i : 0 \leq i \leq n$*, then* $\mathcal{V}[i \mapsto \nu] \overset{def}{=} \nu_0 :: \cdots :: \nu_{i-1} :: \nu :: \nu_{i+1} :: \cdots :: \nu_n$.

Now, instead of updating the kv-store using UpdateKV and then checking for the view shift with vShift as before, we update both the kv-store and the client view at the same time: 1. for each read $(\mathtt{R}, k, v) \in \mathcal{F}$, it adds t to the reader set of the last version of k in u_r (u_r remains unchanged throughout execution); and 2. for each write $(\mathtt{W}, k, v)$, it appends a new version $(v, t, \emptyset)$ to $\mathcal{K}(k)$ as well as add the position of the new version $|\mathcal{K}(k)|$ to the new view u'_w.

We show the the effect of applying a read operation $(\mathtt{R}, k, v)$ and a write operation $(\mathtt{W}, k, v)$ in figs. 4.1a and 4.1b respectively. Notice the difference of the final view in fig. 4.1b (highlighted in green) compared to the view in UpdateKV in fig. 3.3b.

The function NewUpdateKV is well-defined in that for any $\mathcal{K}, u_r, u_w, \mathcal{F}, t, cl$ such that $\mathcal{K}$ and $\mathcal{F}$ are well-formed, $u_r, u_w \in \textsc{Views}(\mathcal{K})$ and $t = \mathsf{FreshTxid}(\mathcal{K}, cl)$, the resulting kv-store and the resulting view $\mathsf{NewUpdateKV}(\mathcal{K}, u_r, u_w, \mathcal{F}, t)$ are a well-formed kv-store and a well-formed view, and are uniquely defined.

We give a reduction semantics of NewUpdateKV and prove the well-formedness of the resulting kv-store-view pair by induction on the length of the evaluation path. We prove the uniqueness of the kv-store and view using a confluence argument.

Definition 4.5 (NewUpdateKV semantics)**.** *The small-step semantics for* NewUpdateKV *has a judgment of the form*

$$u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow_n (\mathcal{K}', u'_w, \emptyset) \textit{ for } |\mathcal{F}| = n$$

and is defined inductively by

$$\begin{aligned}
&u_r, t \vdash (\mathcal{K}, u_w, \emptyset) \rightarrow_0 (\mathcal{K}, u_w, \emptyset)\\
&u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow_{n+1} (\mathcal{K}', u'_w, \emptyset) \text{ if } \exists o \in \mathcal{F}, \mathcal{K}'', u''_w. \text{ such that}\\
&\qquad u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow (\mathcal{K}'', u''_w, \mathcal{F} \setminus \{o\}) \text{ and}\\
&\qquad u_r, t \vdash (\mathcal{K}'', u''_w, \mathcal{F} \setminus \{o\}) \rightarrow_n (\mathcal{K}', u'_w, \emptyset)
\end{aligned}$$

where the relation $\rightarrow$ is defined by

$$\begin{aligned}
u_r, t \vdash (\mathcal{K}, u_w, \{(\mathtt{R}, k, v)\} \uplus \mathcal{F}) \rightarrow (\mathcal{K}', u_w, \mathcal{F}) \text{ where } & i = \max_<(u_r(k)),\ (v, t', T) = \mathcal{K}(k, i) \text{ and}\\
& \mathcal{K}' = \mathcal{K}[k \mapsto \mathcal{K}(k)[i \mapsto (v, t', T \uplus \{t\})]]
\end{aligned}$$

$$\begin{aligned}
u_r, t \vdash (\mathcal{K}, u_w, \{(\mathtt{W}, k, v)\} \uplus \mathcal{F}) \rightarrow (\mathcal{K}', u'_w, \mathcal{F}) \text{ where } & u'_w = u_w[k \mapsto u_w(k) \uplus \{|\mathcal{K}(k)|\}] \text{ and}\\
& \mathcal{K}' = \mathcal{K}[k \mapsto \mathcal{K}(k) :: (v, t, \emptyset)]
\end{aligned}$$

We write $\mathsf{NewUpdateKV}(\mathcal{K}, u_r, u_w, \mathcal{F}, t) = (\mathcal{K}', u'_w) \Leftrightarrow u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow_{|\mathcal{F}|} (\mathcal{K}', u'_w, \emptyset)$.

Proposition 4.6 (Well-formed NewUpdateKV result)**.** *Given a well-formed kv-store $\mathcal{K} \in$ KVS, views on the kv-store $u_r, u_w \in \textsc{Views}(\mathcal{K})$, a well-formed fingerprint $\mathcal{F} \in$ Fp, and a fresh transaction identifier $t = \mathsf{FreshTxid}(\mathcal{K}, cl)$ for a client cl, let $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow_n (\mathcal{K}', u'_w, \emptyset)$ for $|\mathcal{F}| = n$, then the updated kv-store $\mathcal{K}'$ is well-formed, and the updated view u'_w is well-formed with respect to $\mathcal{K}'$.*

Proof sketch. Intuitively, the fingerprint $\mathcal{F}$ contains at most one read and one write per key, so the resulting kv-store must be uniquely defined. The fresh transaction identifier t is strictly greater than any existing identifiers in $\mathcal{K}$ for the same client cl, and the view u_r is a well-formed view on the kv-store $\mathcal{K}$, so the resulting kv-store must be well-formed. Similarly, since each write operation in $\mathcal{F}$ adds the new version to the write-view, and the original write-view u_w is well-formed, then the resulting view must be well-formed. □

Proof. We prove proposition 4.6 by showing the following result

$$\begin{aligned}
&\mathtt{WfKvs}(\mathcal{K}) \wedge u_r \in \textsc{Views}(\mathcal{K}) \wedge u_w \in \textsc{Views}(\mathcal{K}) \wedge\\
&\forall t' \in \mathcal{K}.\, (t, t') \notin \mathsf{SO} \wedge \forall k \in \textsc{Key}.\, \forall v \in \textsc{Value}.\, \forall cl \in \textsc{ClientID}.\\
&\qquad \big((\mathtt{R}, k, v) \in \mathcal{F} \Rightarrow \forall i'.\, t \notin \mathsf{rs}(\mathcal{K}(k, i'))\big)\\
&\qquad \wedge \big((\mathtt{W}, k, v) \in \mathcal{F} \Rightarrow \forall i'.\, t \neq \mathsf{w}(\mathcal{K}(k, i'))\big)\\
&\qquad \wedge \forall i \in u(k)\,.\, t \neq \mathsf{w}(\mathcal{K}(k, i))\\
&\qquad \Rightarrow \mathtt{WfKvs}(\mathcal{K}') \wedge u_r \in \textsc{Views}(\mathcal{K}') \wedge u'_w \in \textsc{Views}(\mathcal{K}')
\end{aligned} \tag{4.1}$$

We prove eq. (4.1) by induction on n.

1. **Base Case** $n = 0$. By definition 4.5 $u_w = u'_w$ and $\mathcal{K} = \mathcal{K}'$. Therefore $\texttt{WfKvs}\,(\mathcal{K}') \wedge u_r \in \textsc{Views}\,(\mathcal{K}') \wedge u'_w \in \textsc{Views}\,(\mathcal{K}')$ as required.

2. **Inductive Case** $n = k + 1$. Now $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow_{k+1} (\mathcal{K}', u'_w, \emptyset)$.

   By definition 4.5 there exist $o \in \mathcal{F}, \mathcal{K}'', u''_w$ such that $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow (\mathcal{K}'', u''_w, \mathcal{F} \setminus \{o\})$ and $u_r, t \vdash (\mathcal{K}'', u''_w, \mathcal{F} \setminus \{o\}) \rightarrow_k (\mathcal{K}', u'_w, \emptyset)$. Let $\mathcal{F}' = \mathcal{F} \setminus \{o\}$.

   The operation o can be either a read or a write.

   (a) **Case** $o = (\texttt{R}, k, v)$. The write-view u_w is unchanged $u''_w = u_w$. Let index $i = \mathsf{Max}_<\,(u_r\,(k))$, old version $(v, t', T) = \mathcal{K}\,(k, i)$, and new version list $\mathcal{V} = \mathcal{K}\,(k)\,[i \mapsto (v, t', T \uplus \{t\})]$. The intermediate kv-store $\mathcal{K}''$ is defined by $\mathcal{K}'' = \mathcal{K}\,[k \mapsto \mathcal{V}]$.

   Since the original kv-store $\mathcal{K}$ satisfies eq. (4.1), the fingerprint has at most one read per key, the fresh transaction identifier $t \notin \mathsf{rs}(\mathcal{K}\,(k, i'))$ for all i' such that $0 \leq i' < |\mathcal{K}\,(k)|$, and therefore $\mathcal{K}''$ satisfies the well-formed condition eq. (3.2) in definition 3.4.

   Because $(t, t') \notin \mathsf{SO}$ for any $t' \in \mathcal{K}$, and $t \neq \mathsf{w}(\mathcal{K}\,(k'', i''))$ for k'', i'' such that $i'' \in u_r\,(k'')$, it cannot be that $(t', t) \in \mathsf{SO}$ for some $t' = \mathsf{w}(\mathcal{K}\,(k, i'))$ where $i' \in u_r\,(k)$. Therefore $\mathcal{K}''$ satisfies eq. (3.3).

   eqs. (3.1) and (3.4) are trivially true for $\mathcal{K}''$. We have shown that the intermediate kv-store $\mathcal{K}''$ is well-formed, $\texttt{WfKvs}\,(\mathcal{K}'')$.

   It is easy to see that both eqs. (3.5) and (3.6) hold for the read- and write-view with respect to $\mathcal{K}''$ giving $u_r \in \textsc{Views}\,(\mathcal{K}'') \wedge u''_w \in \textsc{Views}\,(\mathcal{K}'')$.

   Since $\mathcal{F}$ is well-formed, it follows that $(\texttt{R}, k, v') \notin \mathcal{F}'$, which means that $u_r, t, \mathcal{K}'', u''_w, \mathcal{F}'$ satisfy the invariant eq. (4.1).

   By the inductive hypothesis the final kv-store $\mathcal{K}'$ is well-formed, and the updated view u'_w is well-formed with respect to $\mathcal{K}'$.

   (b) **Case** $o = (\texttt{W}, k, v)$. Let new version list $\mathcal{V} = \mathcal{K}\,(k) :: [(v, t, \emptyset)]$. The intermediate kv-store $\mathcal{K}''$ is defined by $\mathcal{K}'' = \mathcal{K}\,[k \mapsto \mathcal{V}]$.

   Since the original kv-store $\mathcal{K}$ satisfies eq. (4.1), the fingerprint has at most one write per key, the fresh identifier $t \neq \mathsf{w}(\mathcal{K}\,(k, i'))$ for all i' such that $0 \leq i' < |\mathcal{K}\,(k)|$, and therefore $\mathcal{K}''$ satisfies eq. (3.2).

   Because $(t, t') \notin \mathsf{SO}$ for any $t' \in \mathcal{K}$, and t wrote the last version for k in $\mathcal{K}''$, it cannot be that $(t', t) \in \mathsf{SO}$ for some t' in $\mathcal{K}\,(k)$. Therefore $\mathcal{K}''$ satisfies eq. (3.4).

   eqs. (3.1) and (3.3) are trivially true for $\mathcal{K}''$. We have shown that the intermediate kv-store is well-formed, $\texttt{WfKvs}\,(\mathcal{K}'')$.

   The intermediate write-view is defined by $u''_w = u_w\,[k \mapsto u_w\,(k) \uplus \{|\mathcal{K}\,(k)|\}]$. Since $|\mathcal{K}''| = |\mathcal{K}| + 1$, eq. (3.5) holds for u''_w on $\mathcal{K}''$. Since $|\mathcal{K}|$ is added to $u''_w(k)$ for every write operation in $\mathcal{F}$, eq. (3.6) also holds for u''_w on $\mathcal{K}''$ giving $u''_w \in \textsc{Views}\,(\mathcal{K}'')$. It is clear that $u_r \in \textsc{Views}\,(\mathcal{K}'')$.

$$\begin{tikzcd}
{} \arrow[r, "o_1"] \arrow[d, "o_2"'] & {} \arrow[d, "o_2"] \\
{} \arrow[r, "o_1"] & {}
\end{tikzcd}$$

Figure 4.2: Single step confluence (lemma 4.7)

As $\mathcal{F}$ is well-formed, it follows that $(\mathtt{W}, k, v') \notin \mathcal{F}'$, which means that $u_r, t, \mathcal{K}'', u''_w, \mathcal{F}'$ satisfy the invariant eq. (4.1).

By the inductive hypothesis the final kv-store $\mathcal{K}'$ is well-formed, and the updated view u'_w is well-formed with respect to $\mathcal{K}'$. □

We prove the uniqueness of the resulting kv-store-view pair using a confluence argument.

Lemma 4.7 (Single step confluence). *Let $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F} \uplus \{o_1, o_2\}) \to (\mathcal{K}', u'_w, \mathcal{F} \uplus \{o_2\})$ and $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F} \uplus \{o_1, o_2\}) \to (\mathcal{K}'', u''_w, \mathcal{F} \uplus \{o_1\})$ then $\exists \mathcal{K}^o, u^o_w$. such that $u_r, t \vdash (\mathcal{K}', u'_w, \mathcal{F} \uplus \{o_2\}) \to (\mathcal{K}^o, u^o_w, \mathcal{F})$ and $u_r, t \vdash (\mathcal{K}'', u''_w, \mathcal{F} \uplus \{o_1\}) \to (\mathcal{K}^o, u^o_w, \mathcal{F})$.*

Proof sketch. We have four cases to conside: read-read, write-write, read-write, and write-read. Since the fingerprint $\mathcal{F}$ contains at most one read and one write per key, the order of application of the read-read and write-write cases does not matter and we get the same kv-store and view. For the other two cases the keys might be the same. Since reads are done from the read-view u_r that remains unchanged, the index of the version to which we add t as a reader is the same regardless of the order of application, and satisfies the last-write-wins condition. Only the write operation modifies the write-view, hence the resulting view is the same. The write-read case is similar to the read-write case. □

Proof. Let fingerprint $\mathcal{F}' = \mathcal{F} \uplus \{o_1, o_2\}$. There are four cases to consider:

1. **Case** $o_1 = (\mathtt{R}, k, v)$ and $o_2 = (\mathtt{R}, k', v')$. By definition 3.8, $\mathcal{F}'$ contains at most one read per key so the keys must be different $k \neq k'$. Let indices $i = \mathsf{Max}_<(u_r(k))$ and $i' = \mathsf{Max}_<(u_r(k'))$, versions $(v, t', T) = \mathcal{K}(k, i)$ and $(v', t'', T') = \mathcal{K}(k', i')$. Let the new version lists $\mathcal{V} = \mathcal{K}(k)[i \mapsto (v, t', T \cup \{t\})]$ and $\mathcal{V}' = \mathcal{K}(k')[i \mapsto (v', t'', T' \cup \{t\})]$. Neither operation affects u_w.

   We have

   $$u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F} \uplus \{o_1, o_2\}) \to (\mathcal{K}[k \mapsto \mathcal{V}], u_w, \mathcal{F} \uplus \{o_2\})$$

   and

   $$u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F} \uplus \{o_1, o_2\}) \to (\mathcal{K}[k' \mapsto \mathcal{V}'], u_w, \mathcal{F} \uplus \{o_1\}).$$

We get

$$u_r, t \vdash (\mathcal{K}[k \mapsto \mathcal{V}], u_w, \mathcal{F} \uplus \{o_2\}) \rightarrow \\ (\mathcal{K}[k \mapsto \mathcal{V}][k' \mapsto \mathcal{V}'], u_w, \mathcal{F})$$

and

$$u_r, t \vdash (\mathcal{K}[k' \mapsto \mathcal{V}'], u_w, \mathcal{F} \uplus \{o_1\}) \rightarrow \\ (\mathcal{K}[k' \mapsto \mathcal{V}'][k \mapsto \mathcal{V}], u_w, \mathcal{F}).$$

Since $k \neq k'$, it is easy to see that $\mathcal{K}[k \mapsto \mathcal{V}][k' \mapsto \mathcal{V}'] = \mathcal{K}[k' \mapsto \mathcal{V}'][k \mapsto \mathcal{V}]$. These imply lemma 4.7.

2. **Case** $o_1 = (\mathtt{W}, k, v)$ and $o_2 = (\mathtt{W}, k', v')$. By definition 3.8, $\mathcal{F}'$ contains at most one write per key so the keys must be different $k \neq k'$. Let version lists $\mathcal{V} = \mathcal{K}(k) :: (v, t, \emptyset)$ and $\mathcal{V}' = \mathcal{K}(k') :: (v', t, \emptyset)$. Let sets $S_k = u_w(k) \uplus \{|\mathcal{K}(k)|\}$ and $S_{k'} = u_w(k') \uplus \{|\mathcal{K}(k')|\}$.

   We have

$$u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F} \uplus \{o_1, o_2\}) \rightarrow \\ (\mathcal{K}[k \mapsto \mathcal{V}], u_w[k \mapsto S_k], \mathcal{F} \uplus \{o_2\})$$

   and

$$u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F} \uplus \{o_1, o_2\}) \rightarrow \\ (\mathcal{K}[k' \mapsto \mathcal{V}'], u_w[k' \mapsto S_{k'}], \mathcal{F} \uplus \{o_1\}).$$

   We get

$$u_r, t \vdash (\mathcal{K}[k \mapsto \mathcal{V}], u_w[k \mapsto S_k], \mathcal{F} \uplus \{o_2\}) \rightarrow \\ (\mathcal{K}[k \mapsto \mathcal{V}][k' \mapsto \mathcal{V}'], u_w[k \mapsto S_k][k' \mapsto S_{k'}], \mathcal{F})$$

   and

$$u_r, t \vdash (\mathcal{K}[k' \mapsto \mathcal{V}'], u_w[k' \mapsto S_{k'}], \mathcal{F} \uplus \{o_1\}) \rightarrow \\ (\mathcal{K}[k' \mapsto \mathcal{V}'][k \mapsto \mathcal{V}], u_w[k' \mapsto S_{k'}][k \mapsto S_k], \mathcal{F}).$$

   Since $k \neq k'$, it is clear that

$$\mathcal{K}[k \mapsto \mathcal{V}][k' \mapsto \mathcal{V}'] = \\ \mathcal{K}[k' \mapsto \mathcal{V}'][k \mapsto \mathcal{V}].$$

   It is also the case that

$$u_w[k \mapsto S_k][k' \mapsto S_{k'}] = \\ u_w[k' \mapsto S_{k'}][k \mapsto S_k].$$

   These imply lemma 4.7.

3. **Case** $o_1 = (\mathtt{R}, k, v)$ and $o_2 = (\mathtt{W}, k', v')$. In this case k and k' may be the same key.

   (a) **Case** $k \neq k'$. Let index $i = \mathsf{Max}_<(u_r(k))$ and version $(v, t', T) = \mathcal{K}(k, i)$. Let the new version list $\mathcal{V} = \mathcal{K}(k)[i \mapsto (v, t', T \cup \{t\})]$. Because $k \neq k'$,

$$\mathcal{K}[k \mapsto \mathcal{V}][k' \mapsto \mathcal{K}(k') :: (v', t, \emptyset)] = \mathcal{K}[k' \mapsto \mathcal{K}(k') :: (v', t, \emptyset)][k \mapsto \mathcal{V}].$$

   For the views, only the write operation updates u_w,

$$(u_w)[k' \mapsto u_w(k') \uplus \{|\mathcal{K}(k')|\}] = (u_w[k' \mapsto u_w(k') \uplus \{|\mathcal{K}(k')|\}]).$$

$$\begin{tikzcd}
{} \arrow[r, "o_1"] \arrow[d, "o"'] & {} \arrow[r, "o_2"] \arrow[d, "o"'] & {} \arrow[r, dashed] \arrow[d, "o"'] & {} \arrow[r, "o_i = o"] \arrow[d, "o"'] & {} \arrow[r, "o_{i+1}"] \arrow[dl, dashed, no head] & {} \arrow[r, dashed] & {} \arrow[r, "o_{n+1}"] & {} \\
{} \arrow[r, "o_1"'] & {} \arrow[r, "o_2"'] & {} \arrow[r, dashed] & {} \arrow[r, "o_{i+1}"'] & {} \arrow[r, dashed] & {} \arrow[r, "o_{n+1}"'] & {} &
\end{tikzcd}$$

Figure 4.3: One side confluence (lemma 4.8)

(b) **Case** $k = k'$. Let index $i = \mathsf{Max}_<(u_r(k))$, version list $\mathcal{V} = \mathcal{K}(k)$ and version $(v, t', T) = \mathcal{V}(i)$. Since $u_r \in \textsc{Views}(\mathcal{K})$ and remains unchanged, the index i must be in bound, that is, $0 \le i < |\mathcal{K}(k)|$, therefore

$$\big(\mathcal{V}[i \mapsto (v, t', T \cup \{t\})]\big) :: (v', t, \emptyset) = \big(\mathcal{V} :: (v', t, \emptyset)\big)[i \mapsto (v, t', T \cup \{t\})].$$

The argument for views is the same as when the keys are different.

Both cases imply lemma 4.7.

4. **Case** $o = (\mathtt{W}, k, v)$ and $o' = (\mathtt{R}, k', v')$. This case is the same as item 3. □

Lemma 4.8 (One side confluence). *Let $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow_{n+1} (\mathcal{K}', u'_w, \emptyset)$ for $|\mathcal{F}| = n + 1$ and let $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow (\mathcal{K}'', u''_w, \mathcal{F} \setminus \{o\})$ then $u_r, t \vdash (\mathcal{K}'', u''_w, \mathcal{F} \setminus \{o\}) \rightarrow_n (\mathcal{K}', u'_w, \emptyset)$.*

Proof. We prove lemma 4.8 by induction on n.

1. **Base Case** $n = 0$. The fingerprint size $|\mathcal{F}| = 1$ so it must be that $\mathcal{F} = \{o\}$. We have $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow_1 (\mathcal{K}', u'_w, \emptyset)$. By definition 4.5 there exist $o' \in \mathcal{F}, \mathcal{K}^o, u^o_w$ such that $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow (\mathcal{K}^o, u^o_w, \mathcal{F} \setminus \{o'\})$ and $u_r, t \vdash (\mathcal{K}^o, u^o_w, \mathcal{F} \setminus \{o'\}) \rightarrow_0 (\mathcal{K}', u'_w, \emptyset)$. But $o = o'$ and we have that $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow (\mathcal{K}'', u''_w, \mathcal{F} \setminus \{o\})$ so $\mathcal{K}'' = \mathcal{K}^o$ and $u''_w = u^o_w$. Substituting in we get $u_r, t \vdash (\mathcal{K}'', u''_w, \mathcal{F} \setminus \{o\}) \rightarrow_0 (\mathcal{K}', u'_w, \emptyset)$ as required.

2. **Inductive Case** $n = k + 1$. We have that $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow_{(k+1)+1} (\mathcal{K}', u'_w, \emptyset)$ and $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow (\mathcal{K}'', u''_w, \mathcal{F} \setminus \{o\})$.

   By definition 4.5 there exists $o^1 \in \mathcal{F}, \mathcal{K}^1, u^1_w$ such that $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow (\mathcal{K}^1, u^1_w, \mathcal{F} \setminus \{o^1\})$ and $u_r, t \vdash (\mathcal{K}^1, u^1_w, \mathcal{F} \setminus \{o^1\}) \rightarrow_{k+1} (\mathcal{K}', u'_w, \emptyset)$.

   By lemma 4.7 there exists $\mathcal{K}^o, u^o_w$ such that $u_r, t \vdash (\mathcal{K}^1, u^1_w, \mathcal{F} \setminus \{o^1\}) \rightarrow (\mathcal{K}^o, u^o_w, \mathcal{F} \setminus \{o^1, o\})$ and $u_r, t \vdash (\mathcal{K}'', u''_w, \mathcal{F} \setminus \{o\}) \rightarrow (\mathcal{K}^o, u^o_w, \mathcal{F} \setminus \{o^1, o\})$.

   From this and the inductive hypothesis we get $u_r, t \vdash (\mathcal{K}'', u''_w, \mathcal{F} \setminus \{o\}) \rightarrow_{k+1} (\mathcal{K}', u'_w, \emptyset)$ as required. □

Lemma 4.9 (Confluence). *Let $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow_n (\mathcal{K}', u'_w, \emptyset)$ and $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow_n (\mathcal{K}'', u''_w, \emptyset)$ then $\mathcal{K}' = \mathcal{K}''$ and $u'_w = u''_w$.*

Proof. We prove lemma 4.9 by induction on n.

1. **Base Case** $n = 0$. We have $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow_0 (\mathcal{K}', u'_w, \emptyset)$ and $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow_0 (\mathcal{K}'', u''_w, \emptyset)$. By definition 4.5 $\mathcal{K} = \mathcal{K}'$, $\mathcal{K} = \mathcal{K}''$ and $u_w = u'_w$, $u_w = u''_w$. We get $\mathcal{K}' = \mathcal{K}''$ and $u'_w = u''_w$ as required.
2. **Inductive Case** $n = k + 1$. We have $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow_{k+1} (\mathcal{K}', u'_w, \emptyset)$. By definition 4.5 there exists $o \in \mathcal{F}, \mathcal{K}^o, u^o_w$ such that $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow (\mathcal{K}^o, u^o_w, \mathcal{F} \setminus \{o\})$ and $u_r, t \vdash (\mathcal{K}^o, u^o_w, \mathcal{F} \setminus \{o\}) \rightarrow_k (\mathcal{K}', u'_w, \emptyset)$.

   We also have $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow_{k+1} (\mathcal{K}'', u''_w, \emptyset)$. By lemma 4.8 $u_r, t \vdash (\mathcal{K}^o, u^o_w, \mathcal{F} \setminus \{o\}) \rightarrow_k (\mathcal{K}'', u''_w, \emptyset)$. $\mathcal{K}' = \mathcal{K}''$ and $u'_w = u''_w$ by the inductive hypothesis as required. □

Theorem 4.10 (Well-defined NewUpdateKV). *Given a well-formed kv-store $\mathcal{K} \in$ KVS, views on the kv-store $u_r, u_w \in$ VIEWS $(\mathcal{K})$, a well-formed fingerprint $\mathcal{F} \in$ FP and a fresh transaction identifier $t =$ FreshTxid $(\mathcal{K}, cl)$ for a client cl,*

let $(\mathcal{K}', u'_w) =$ NewUpdateKV $(\mathcal{K}, u_r, u_w, \mathcal{F}, t)$, then the pair $(\mathcal{K}', u'_w)$ is uniquely defined, where $\mathcal{K}'$ is a well-formed kv-store, and u'_w is a well-formed view with respect to $\mathcal{K}'$.

Theorem 4.10 is the result of putting together our well-formdness result of the function NewUpdateKV (proposition 4.6) and the uniqueness result that follows from the confluence property shown in lemma 4.9.

We prove that the resulting kv-store from our definition of NewUpdateKV is equal to that returned from the original transactional update function UpdateKV. Similarly to definition 4.5, we give a reduction semantics for UpdateKV.

Definition 4.11 (UpdateKV semantics). *The small-step semantics for* UpdateKV *has a judgment of the form*

$$u, t \vdash (\mathcal{K}, \mathcal{F}) \rightarrow_n (\mathcal{K}', \emptyset) \textit{ for } |\mathcal{F}| = n$$

and is defined inductively by

$$\begin{aligned}
&u, t \vdash (\mathcal{K}, \emptyset) \rightarrow_0 (\mathcal{K}, \emptyset)\\
&u, t \vdash (\mathcal{K}, \mathcal{F}) \rightarrow_{n+1} (\mathcal{K}', \emptyset) \textit{ if } \exists o \in \mathcal{F}, \mathcal{K}''. \textit{ such that}\\
&\qquad\qquad u, t \vdash (\mathcal{K}, \mathcal{F}) \rightarrow (\mathcal{K}'', \mathcal{F} \setminus \{o\}) \textit{ and}\\
&\qquad\qquad u, t \vdash (\mathcal{K}'', \mathcal{F} \setminus \{o\}) \rightarrow_n (\mathcal{K}', \emptyset)
\end{aligned}$$

where the relation $\rightarrow$ is defined by

$$u, t \vdash (\mathcal{K}, \{(\mathtt{R}, k, v)\} \uplus \mathcal{F}) \rightarrow (\mathcal{K}', \mathcal{F}) \textit{ where } i = \max_<(u(k)),\ (v, t', T) = \mathcal{K}(k, i) \textit{ and}$$
$$\mathcal{K}' = \mathcal{K}[k \mapsto \mathcal{K}(k)[i \mapsto (v, t', T \uplus \{t\})]]$$

$$u, t \vdash (\mathcal{K}, \{(\mathtt{W}, k, v)\} \uplus \mathcal{F}) \rightarrow (\mathcal{K}', \mathcal{F}) \textit{ where } \mathcal{K}' = \mathcal{K}[k \mapsto \mathcal{K}(k) :: (v, t, \emptyset)]$$

We write $\mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t) = \mathcal{K}' \Leftrightarrow u, t \vdash (\mathcal{K}, \mathcal{F}) \rightarrow_{|\mathcal{F}|} (\mathcal{K}', \emptyset)$.

Theorem 4.12 (Well-defined UpdateKV). *Given a well-formed kv-store $\mathcal{K} \in \mathrm{KVS}$, a view on the kv-store $u \in \textsc{Views}(\mathcal{K})$, a well-formed fingerprint $\mathcal{F} \in \textsc{Fp}$ and a fresh transaction identifier $t \in \mathsf{NextTxID}(\mathcal{K}, cl)$ for a client cl, the new kv-store $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t)$ is a uniquely defined and well-formed kv-store.*

Proof of theorem 4.12 is given in [1].

The lemmas analogous to lemma 4.7, lemma 4.8, and lemma 4.9 hold for UpdateKV.

Proposition 4.13 (NewUpdateKV kv-store equals UpdateKV). *Given a kv-store $\mathcal{K} \in \mathrm{KVS}$, views $u_r, u_w \in \textsc{Views}(\mathcal{K})$, a fingerprint $\mathcal{F} \in \textsc{Fp}$, and a transaction identifier $t \in \textsc{TxID}_0$,*

Let $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow_n (\mathcal{K}', u'_w, \emptyset)$ and $u_r, t \vdash (\mathcal{K}, \mathcal{F}) \rightarrow_n (\mathcal{K}'', \emptyset)$ then $\mathcal{K}' = \mathcal{K}''$.

Proof sketch. Intuitively, since all read operations are done using u_r, which remains unchanged throughout execution, the effect of reads is the same. Since u_w is never read, the addition of new versions written by t to the write-view in NewUpdateKV does not affect the kv-store. Since there is at most one write per key in $\mathcal{F}$, the effect of write operations on the kv-store under both functions is the same. Therefore, the resulting kv-stores are equal. □

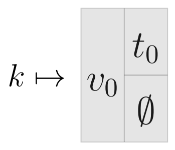
Proof. We prove proposition 4.13 by induction on n.

1. **Base Case** $n = 0$. By definition 4.5 $\mathcal{K}' = \mathcal{K}$. By definition 4.11 $\mathcal{K}'' = \mathcal{K}$. We get $\mathcal{K}' = \mathcal{K}''$ as required.

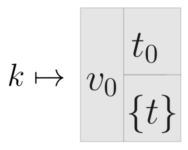
2. **Inductive Case** $n = k + 1$. By definition 4.5 there exist $o \in \mathcal{F}, \mathcal{K}^o, u^o_w$ such that $u_r, t \vdash (\mathcal{K}, u_w, \mathcal{F}) \rightarrow (\mathcal{K}^o, u^o_w, \mathcal{F} \backslash \{o\})$ and $u_r, t \vdash (\mathcal{K}^o, u^o_w, \mathcal{F} \backslash \{o\}) \rightarrow_k (\mathcal{K}', u'_w, \emptyset)$.
   By definition 4.11 there exist $o' \in \mathcal{F}, \mathcal{K}^{o'}$ such that $u_r, t \vdash (\mathcal{K}, \mathcal{F}) \rightarrow (\mathcal{K}^{o'}, \mathcal{F} \setminus \{o'\})$ and $u_r, t \vdash (\mathcal{K}o', \mathcal{F} \setminus \{o'\}) \rightarrow_k (\mathcal{K}'', \emptyset)$.

   We have shown that for NewUpdateKV operations can be applied in any order. A similar result for UpdateKV is shown in [1]. Therefore, it is enough to show that applying the operations in the same order results in equal kv-stores $o = o'$. The next operation o can be either a read or a write operation.

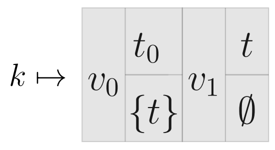
   (a) **Case** $o = (\mathtt{R}, k, v)$. Since no step modifies u_r in either function, the index $i = \mathsf{Max}_<(u_r(k))$ is the same for both functions. Let version $(v, t', T) = \mathcal{K}(k, i)$ and new version list $\mathcal{V} = \mathcal{K}(k)[i \mapsto (v, t', T \uplus \{t\})]$. It is easy to see that the intermediate kv-stores are both defined by $\mathcal{K}^o = \mathcal{K}[k \mapsto \mathcal{V}] = \mathcal{K}^{o'}$ as required.



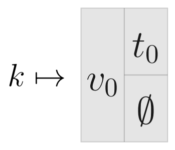
(a) Initial kv-store and view



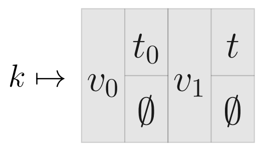
(b) Kv-store and view, read first



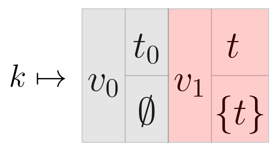
(c) Kv-store and view, read followed by write



(d) Initial kv-store and view



(e) Kv-store and view, write first



(f) Kv-store and view, write followed by read

Figure 4.4: Wrong update: the effect of updating a single view on a kv-store in transactional updates

(b) **Case** $o = (\mathtt{W}, k, v)$. Since the write-view u_w is never read, modifications to it do not affect the resulting kv-store. Let new version list $\mathcal{V} = \mathcal{K}(k) :: (v, t, \emptyset)$. The intermediate kv-stores are both defined by $\mathcal{K}^o = \mathcal{K}[k \mapsto \mathcal{V}] = \mathcal{K}^{o'}$ as required. $\square$

The reason for having two views, a read-view u_r and a write-view u_w, is to allow operations from the fingerprint to be chosen in any order in a way similar to the original UpdateKV definition described in section 3.2. We demonstrate the need for two views with an example.

Example 4.14. *Consider the following incorrect definition using a single view:*

$$
\begin{aligned}
\mathsf{WrongUpdateKV}(\mathcal{K}, u, \emptyset, t) &\overset{def}{=} (\mathcal{K}, u) \\
\mathsf{WrongUpdateKV}(\mathcal{K}, u, \{(\mathtt{R}, k, v)\} \uplus \mathcal{F}, t) &\overset{def}{=} \\
&\mathtt{let}\ i = \max{}_<(u(k))\ \mathtt{and}\ (v, t', T) = \mathcal{K}(k, i)\ \mathtt{in} \\
&\quad \mathsf{WrongUpdateKV}(\mathcal{K}[k \mapsto \mathcal{K}(k)[i \mapsto (v, t', T \uplus \{t\})]], u, \mathcal{F}, t) \\
\mathsf{WrongUpdateKV}(\mathcal{K}, u, \{(\mathtt{W}, k, v)\} \uplus \mathcal{F}, t) &\overset{def}{=} \mathtt{let}\ \mathcal{K}' = \mathcal{K}[k \mapsto \mathcal{K}(k) :: (v, t, \emptyset)] \\
&\qquad \mathtt{and}\ u' = u[k \mapsto u(k) \uplus \{|\mathcal{K}(k)|\}]\ \mathtt{in} \\
&\qquad\quad \mathsf{WrongUpdateKV}(\mathcal{K}', u', \mathcal{F}, t)
\end{aligned}
$$

Now consider the kv-store presented in fig. 4.4 and a client cl with the initial view u_0. Client cl is executing transaction t with fingerprint $\mathcal{F} = \{(\mathtt{R}, k, v_0), (\mathtt{W}, k, v_1)\}$. We provide a step-by-step breakdown of the call to WrongUpdateKV*, choosing the read operation first followed by the write operation and the other way around. Grey versions are in the view of client cl. As seen in fig. 4.4a and fig. 4.4d, the initial view and kv-store* *are identical. Applying the read first (fig. 4.4b), we add t to the reader set of the latest version in the view - ν_0 with value v_0. Next, we apply the write operation, appending version ν_1 with value v_1 to $\mathcal{K}(k)$ and updating our view (fig. 4.4c). fig. 4.4e shows the updated kv-store and view when applying the write first. Since we update the view as we go to satisfy* `RYW`*, version ν_1 with value v_1 written by t is included in cl's view. There are two possible cases to consider.*

1. ***Case** $v_0 = v_1$.*

   Applying the read operation results in the kv-store and view as shown in fig. 4.4f. We have transaction t in the reader set of a version written by t itself, which is not allowed by the semantics as it violates the well-formedness conditions of a kv-store (eq. (3.3)).

2. ***Case** $v_0 \neq v_1$.*

   The update fails in this case. When applying the read operation, $i = \max_<(u(k))$ sets $i = 1$ and $(v, t', T) = \mathcal{K}(k, i)$ breaks as the values are not equal.

Another possible solution would be to define the function in a way that applies read operations first. Although this would result in the same kv-store-view pair, it would mean that we lose the ability to choose and apply any operation from the fingerprint in any order. Since we would like to keep the implementation close to the original definitions, with a fingerprint being a *set* of opertions, we choose to use the read-view and write-view approach.

Proposition 4.15 (New update satisfies $\mathtt{vShift}_{\mathtt{MR}}$)**.** *Given a kv-store $\mathcal{K} \in \mathrm{KVS}$, a view $u \in \textsc{Views}(\mathcal{K})$, a fingerprint $\mathcal{F} \in \textsc{Fp}$, and a transaction identifier $t \in \textsc{TxID}_0$,*

$$(\mathcal{K}', u') = \mathsf{NewUpdateKV}(\mathcal{K}, u, u, \mathcal{F}, t) \Rightarrow \mathtt{vShift}_{\mathtt{MR}}(\mathcal{K}, u, \mathcal{K}', u').$$

Proof. It is immediate from definition 4.4 that no step removes information from the view. Therefore, the resulting view u' satisfies $u \sqsubseteq u'$ as required. $\square$

Proposition 4.16 (New update satisfies $\mathtt{vShift}_{\mathtt{RYW}}$)**.** *Given a kv-store $\mathcal{K} \in \mathrm{KVS}$, a view $u \in \textsc{Views}(\mathcal{K})$, a fingerprint $\mathcal{F} \in \textsc{Fp}$, and a transaction identifier $t \in \textsc{TxID}_0$,*

$$(\mathcal{K}', u') = \mathsf{NewUpdateKV}(\mathcal{K}, u, u, \mathcal{F}, t) \Rightarrow \mathtt{vShift}_{\mathtt{RYW}}(\mathcal{K}, u, \mathcal{K}', u').$$

Proof. For each $(\mathtt{W}, k, v) \in \mathcal{F}$ we add the length of $\mathcal{K}(k)$ to the write-view u_w, where $\mathcal{K}$ is the kv-store before writing the new version. Clearly, $|\mathcal{K}(k)|$ is the index of the new version, so all versions written by t are in the resulting view u' as required. $\square$

Theorem 4.17 (New update satisfies $\mathtt{vShift}_{\mathtt{MR\cap RYW}}$)**.** *Given a kv-store $\mathcal{K} \in \mathrm{KVS}$, views $u_r, u_w \in \textsc{Views}(\mathcal{K})$, a fingerprint $\mathcal{F} \in \textsc{Fp}$, and a transaction identifier $t \in \textsc{TxID}_0$,*

$$\mathtt{RYW}(cl, \mathcal{K}, u_w) \land (\mathcal{K}', u') = \mathsf{NewUpdateKV}(\mathcal{K}, u_r, u_w, \mathcal{F}, t) \Rightarrow \mathtt{vShift}_{\mathtt{MR\cap RYW}}(\mathcal{K}, u_w, \mathcal{K}', u').$$

Proof. From propositions 4.15 and 4.16 we get $\mathtt{RYW}(cl, \mathcal{K}', u') \land u_w \sqsubseteq u'$ as required. $\square$

Chapter 5

Reformulating the Semantics

In this chapter we use our definitions from chapter 4 to update the semantics of an atomic transaction, previously given in section 3.2. Section 5.1 present an updated CATOMICTRANS rule, which we show to be equivalent to the original rule, and identifies the notion of *minimal views* with respect to an update. In section 5.2 we prove that the RYW and MR properties hold throughout execution, so we can eliminate their checks.

5.1 Redefining CATOMICTRANS

We have shown that using our new transactional update (definition 4.4) results in the same kv-store as that returned by the original UpdateKV function. We have also shown that the view returned by NewUpdateKV satisfies $\texttt{vShift}_{\texttt{MR}\cap\texttt{RYW}}$. In fig. 5.1 we redefine the CATOMICTRANS rule (fig. 3.1) using our new definitions.

In order to show that our new rule is *equivalent* to the original, we need first to define what it means for the rules to be equivalent. With the end goal of using our implementation of the semantics to find litmus tests (section 2.3), our main focus is on the kv-store states obtainable by executing client programs. Therefore, we define the rules as being equivalent if the set of kv-stores reachable by applying transactions using either rule is the same. This is consistent with the definition of consistency models for kv-stores (definition 3.14).

To prove that the rule in fig. 5.1 allows obtaining the same kv-stores, we show that the set of views allowed by using UpdateKV and $\texttt{vShift}_{\texttt{MR}\cap\texttt{RYW}}$ is not restricted when using NewUpdateKV. We do this by showing that the view shift done by NewUpdateKV is *minimal* with respect to $\texttt{vShift}_{\texttt{MR}\cap\texttt{RYW}}$.

Definition 5.1 (Minimal view-shift). *Given kv-stores $\mathcal{K}$, $\mathcal{K}'$, and views over the stores $u \in \text{VIEWS}(\mathcal{K})$ and $u' \in \text{VIEWS}(\mathcal{K}')$, for a client cl, the* minimal *view shift with respect*

$$\textsc{CAtomicTrans}\quad \frac{\begin{array}{c} u \sqsubseteq u'' \quad \sigma = \mathsf{snapshot}(\mathcal{K}, u'') \quad (s, \sigma, \emptyset), \mathtt{T} \rightsquigarrow^* (s', \_, \mathcal{F}), \mathtt{skip} \quad \mathtt{canCommit}_{\mathtt{ET}}(\mathcal{K}, u'', \mathcal{F}) \\ t = \mathsf{FreshTxid}(cl, \mathcal{K}) \qquad (\mathcal{K}', u') = \mathsf{NewUpdateKV}(\mathcal{K}, u'', u'', \mathcal{F}, t) \end{array}}{cl \vdash (\mathcal{K}, u, s), [\mathtt{T}] \xrightarrow{(cl, u'', \mathcal{F})}_{\mathtt{ET}} (\mathcal{K}', u', s'), \mathtt{skip}}$$

Figure 5.1: The updated semantics of an atomic transaction under execution test ET

to $\mathtt{vShift}_{\mathtt{MR}\cap\mathtt{RYW}}$*, written* $\mathtt{vShiftMin}_{\mathtt{MR}\cap\mathtt{RYW}}(\mathcal{K}, u, \mathcal{K}', u')$*, is defined by*

$$\begin{aligned} \mathtt{vShiftMin}_{\mathtt{MR}\cap\mathtt{RYW}}(\mathcal{K}, u, \mathcal{K}', u') \overset{def}{=} & \ \mathtt{vShift}_{\mathtt{MR}\cap\mathtt{RYW}}(\mathcal{K}, u, \mathcal{K}', u') \\ & \wedge \forall u''.\, \mathtt{vShift}_{\mathtt{MR}\cap\mathtt{RYW}}(\mathcal{K}, u, \mathcal{K}', u'') \Rightarrow u' \sqsubseteq u'' \end{aligned}$$

Intuitively, all and only versions written by the committed transaction are added to the new view to satisfy $\mathtt{RYW}(cl, \mathcal{K}', u')$.

Proposition 5.2 (NewUpdateKV minimal view-shift)**.** *Given a kv-store* $\mathcal{K} \in$ KVS*, views* $u_r, u_w \in \textsc{Views}(\mathcal{K})$*, a fingerprint* $\mathcal{F} \in \textsc{Fp}$*, and a transaction identifier* t*,* $(\mathcal{K}', u') = \mathsf{NewUpdateKV}(\mathcal{K}, u, u, \mathcal{F}, t) \Rightarrow \mathtt{vShiftMin}_{\mathtt{MR}\cap\mathtt{RYW}}(\mathcal{K}, u, \mathcal{K}', u')$

Proof. It is easy to see that the view-shift applied by NewUpdateKV (definition 4.4) is minimal. □

We call the view of client cl immediately after t has been committed the *minimal view* of t.

NewUpdateKV is more forward-looking than the old UpdateKV implementation in that it incorporates the minimal view shift into the update. Instead of updating the kv-store and then generating views and eliminating them if they do not satisfy the $\mathtt{vShift}_{\mathtt{MR}\cap\mathtt{RYW}}$ predicate, NewUpdateKV adds the versions written by the executing client to its view during the update. Because the new view has to satisfy RYW, we know that these versions *must* be in all views that satisfy $\mathtt{vShift}_{\mathtt{MR}\cap\mathtt{RYW}}$. By not taking information away from the view, the new view is guaranteed to be ordered after the initial view of the client before executing the update, $u \sqsubseteq u'$. In this way we create the *minimal view shift* required to satisfy the $\mathtt{vShift}_{\mathtt{MR}\cap\mathtt{RYW}}$ predicate.

5.2 Preserving MR and RYW Properties

The vShift predicate of many consistency models is defined by $\mathtt{vShift}_{\mathtt{MR}\cap\mathtt{RYW}}$. We have identified that by updating the client view during a transactional update we can incorporate the minimal view-shift into the update function (NewUpdateKV) and eliminate the vShift check, thus making it more forward.

If we can show that the `RYW` and `MR` properties are preserved throughout execution we can safely eliminate the checks for these properties. Clearly, the initial view u_0 on the initial kv-store $\mathcal{K}_0$ satisfies `RYW` guarantees for every client. There are three points in execution that we need to consider:

1. After applying a transactional update.
2. When advancing the view at the beginning of a transaction while satisfying `canCommit` to incorporate environment information (section 3.2).
3. Between transactions.

5.2.1 Monotonic Read (`MR`) Properties

We have shown that NewUpdateKV satisfies `MR` (proposition 4.15). Trivially, advancing the view satisfies `MR`. Since no information is taken away from the view between transactions, `MR` is always satisfied.

5.2.2 Read Your Writes (`RYW`) Properties

Transactional Update

We show that given a view u that satisfies client read your writes (definition 4.3) with respect to a kv-store $\mathcal{K}$, the resulting view u' from NewUpdateKV satisfies `RYW` with respect to the updated kv-store $\mathcal{K}'$.

Proposition 5.3 (New update preserves `RYW`). *Given a kv-store $\mathcal{K} \in$ KVS, views $u_r, u_w \in \textsc{Views}(\mathcal{K})$, a fingerprint $\mathcal{F} \in \textsc{Fp}$, and a transaction identifier $t \in \textsc{TxID}_0$,*

$$\texttt{RYW}(cl, \mathcal{K}, u_w) \land (\mathcal{K}', u') = \mathsf{NewUpdateKV}(\mathcal{K}, u_r, u_w, \mathcal{F}, t) \Rightarrow \texttt{RYW}(cl, \mathcal{K}', u')$$

where $\texttt{RYW}(cl, \mathcal{K}, u_w)$ *is defined in definition 4.3.*

Proof sketch. Since the original view satisfies $\texttt{RYW}(cl, \mathcal{K}, u_w)$, we know that all previous versions committed by cl are included in the view. Intuitively, for each $(\texttt{W}, k, v) \in \mathcal{F}$ we add the length of $\mathcal{K}(k)$ to the write-view u_w, where $\mathcal{K}$ is the kv-store before writing the new version. Clearly, $|\mathcal{K}(k)|$ is the index of the new version, so all versions written by t are in the resulting view u'. We get that all versions written by transactions of client cl are in the view as required. □

Proof. We prove proposition 5.3 by induction on the size of the fingerprint $\mathcal{F}$:

1. **Base Case** $|\mathcal{F}| = 0$.

   In this case we know that $\mathcal{F} = \emptyset$. proposition 5.3 trivially holds as $u' = u_w$ and $\mathcal{K}' = \mathcal{K}$ by definition 4.4.

2. **Inductive Case** $|\mathcal{F}| > 0$.

   Now $\mathcal{F} = \{o\} \uplus \mathcal{F}'$ for some operation $o \in \textsc{Op}$ and fingerprint $\mathcal{F}' \in \textsc{Fp}$ where o is either a read or a write.

   (a) **Case** $\mathcal{F} = \{(\mathtt{R}, k, v)\} \uplus \mathcal{F}'$. (read operation)
   Let index $i = \mathsf{Max}_{<}(u_r(k))$, old version $(v, t', T) = \mathcal{K}(k, i)$, and new version list $\mathcal{V} = \mathcal{K}(k)[i \mapsto (v, t', T \uplus \{t\})]$. The intermediate kv-store $\mathcal{K}^*$ is defined by $\mathcal{K}^* = \mathcal{K}[k \mapsto \mathcal{V}]$. Since the original kv-store and write-view satisfy $\mathtt{RYW}(cl, \mathcal{K}, u_w)$, and because adding t to the reader set does not influence the antecedent of the inner condition in definition 4.3, the intermediate kv-store and write-view satisfy $\mathtt{RYW}(cl, \mathcal{K}^*, u_w)$ as required.

   (b) **Case** $\mathcal{F} = \{(\mathtt{W}, k, v)\} \uplus \mathcal{F}'$. (write operation)
   Let the new version list $\mathcal{V} = \mathcal{K}(k) :: [(v, t, \emptyset)]$. The intermediate kv-store $\mathcal{K}^*$ is defined by $\mathcal{K}^* = \mathcal{K}[k \mapsto \mathcal{V}]$. Let the intermediate write-view $u_w^* = u_w[k \mapsto u_w(k) \uplus \{|\mathcal{K}(k)|\}]$. Since $\mathtt{RYW}(cl, \mathcal{K}, u_w)$, we only need to consider the difference between the original and intermediate kv-store and write-view. The only difference is the $|\mathcal{K}(k)|^{\text{th}}$ version of key k. Since it is added to u_w^*, the predicate $\mathtt{RYW}(cl, \mathcal{K}^*, u_w)$ holds as required. □

Environment Information

We show that given a view u that satisfies $\mathtt{RYW}(cl, \mathcal{K}, u)$ for a client cl and kv-store $\mathcal{K}$, if we advance the view to $u \sqsubseteq u'$ then the new view u' satisfies $\mathtt{RYW}(cl, \mathcal{K}, u')$.

Proposition 5.4 (Advancing view preserves RYW). *Given a client identifier cl, a kv-store $\mathcal{K}$, and views over the kv-store $u, u' \in \textsc{Views}(\mathcal{K})$, $\mathtt{RYW}(cl, \mathcal{K}, u) \land u \sqsubseteq u' \Rightarrow \mathtt{RYW}(cl, \mathcal{K}, u')$ holds.*

Proof. By definition 3.5 we have that

$$\begin{aligned} u \sqsubseteq u' &\overset{\text{def}}{\Leftrightarrow} \forall k \in \mathrm{dom}(\mathcal{K}).\, u(k) \subseteq u'(k) \\ &\Leftrightarrow \forall k \in \mathrm{dom}(\mathcal{K}).\, \forall i.\, i \in u(k) \Rightarrow i \in u'(k) \end{aligned} \tag{5.1}$$

By definition 4.3 of $\mathtt{RYW}(cl, \mathcal{K}, u)$ we also have that

$$\begin{aligned} &\mathtt{RYW}(cl, \mathcal{K}, u) \overset{def}{=} \\ &\quad \forall n \in \mathbb{N}.\, \forall t_{cl}^n \in \mathtt{TxID}_0.\, \left(\forall k, i.\, \mathsf{w}(\mathcal{K}(k, i)) = t_{cl}^n \Rightarrow i \in u(k)\ \right) \end{aligned} \tag{5.2}$$

From (5.1) and (5.2) we get $\mathtt{RYW}(cl, \mathcal{K}, u')$ as required. □

Between Transaction

Now, suppose that client cl has view u over kv-store $\mathcal{K}$. It is possible that other clients commit transactions to $\mathcal{K}$ between cl's transactions. We define an ordering between kv-stores and prove that updates to $\mathcal{K}$ by other clients do not break the RYW conditions for client cl's view.

Definition 5.5 (Later kv-store)**.** *Given two kv-stores $\mathcal{K}, \mathcal{K}' \in$ KVS, the order between them is defined by:*

$$\begin{aligned}\mathcal{K} \sqsubseteq \mathcal{K}' \overset{\text{def}}{\Leftrightarrow} \forall k \in \operatorname{dom}(\mathcal{K}).\, \forall i \in [0, |\mathcal{K}(k)|).\, & \mathsf{w}(\mathcal{K}(k,i)) = \mathsf{w}(\mathcal{K}'(k,i)) \\ & \wedge \mathsf{val}(\mathcal{K}(k,i)) = \mathsf{val}(\mathcal{K}'(k,i)) \\ & \wedge \mathsf{rs}(\mathcal{K}(k,i)) \subseteq \mathsf{rs}(\mathcal{K}'(k,i))\end{aligned}$$

Intuitively, all versions in $\mathcal{K}$ are also in $\mathcal{K}'$ and potentially have additional readers. Since kv-stores are total functions from keys to lists of versions definition 3.3, we do not need to explicitly specify that the domains are equal, $\operatorname{dom}(\mathcal{K}) = \operatorname{dom}(\mathcal{K}')$. It also follows from the definition that version lists in $\mathcal{K}'$ may have subsequent versions for a key $\forall k \in \operatorname{dom}(\mathcal{K}).\, |\mathcal{K}(k)| \leq |\mathcal{K}'(k)|$.

Proposition 5.6 (Later kv-store preserves RYW)**.** *Given a client identifier cl, kv-stores $\mathcal{K}$, $\mathcal{K}'$, and a view $u \in \textsc{Views}(\mathcal{K})$,*

$$\mathtt{RYW}(cl, \mathcal{K}, u) \wedge \mathcal{K} \sqsubseteq \mathcal{K}' \wedge \forall n.\, t_{cl}^{n} \notin \mathcal{K}' \setminus \mathcal{K} \Rightarrow \mathtt{RYW}(cl, \mathcal{K}', u).$$

Proof. From $u \in \textsc{Views}(\mathcal{K})$ and $\mathcal{K} \sqsubseteq \mathcal{K}'$ we know that $u \in \textsc{Views}(\mathcal{K}')$. From $\forall n.\, t_{cl}^{n} \notin \mathcal{K}' \setminus \mathcal{K}$ and definition 5.5 we get that $\forall n.\, t_{cl}^{n} \in \mathcal{K} \Leftrightarrow t_{cl}^{n} \in \mathcal{K}'$. Combining the above we get $\mathtt{RYW}(cl, \mathcal{K}', u)$ as required. □

Chapter 6

Consistency Models Using NewUpdateKV

Our NewUpdateKV implementation incorporates $\texttt{vShift}_{\texttt{MR}\cap\texttt{RYW}}$ into a transactional update, updating the client view during the update of the kv-store by adding the versions written by the executing client to its view, as a step towards a more forward-looking model.

In section 6.1 we discuss the effect of using the new transactional update definition on consistency models whose `vShift` predicate is defined by $\texttt{vShift}_{\texttt{MR}\cap\texttt{RYW}}$ (fig. 3.2) such as Causal Consistency (`CC`), Snapshot Isolation (`SI`), and Parallel Snapshot Isolation (`PSI`) . We give the example of the Update Atomic (`UA`) consistency model as one with a different view-shift condition (fig. 3.2) and compare the outcomes of using NewUpdateKV on such model in section 6.2.

6.1 Models with $\texttt{vShift}_{\texttt{MR}\cup\texttt{RYW}}$

Apart from the improvement gained by not having to check the `vShift` condition when using NewUpdateKV, models with the $\texttt{vShift}_{\texttt{MR}\cap\texttt{RYW}}$ view shift condition are uninfluenced by our new definition.

As shown by proposition 4.13, the resulting kv-store from NewUpdateKV is the same as that returned by UpdateKV. We have also shown that the view-shift applied by $(\mathcal{K}', u') = \mathsf{NewUpdateKV}\,(\mathcal{K}, u, u, \mathcal{F}, t)$ is minimal, $\texttt{vShiftMin}_{\texttt{MR}\cap\texttt{RYW}}\,(\mathcal{K}, u, \mathcal{K}', u')$ in proposition 5.2. This means that any view u'' that satisfies $\texttt{vShift}_{\texttt{MR}\cap\texttt{RYW}}\,(\mathcal{K}, u, \mathcal{K}', u'')$ must be more advanced $u' \sqsubseteq u''$. Since a client's view may be advanced at the beginning of a new transaction before obtaining a snapshot, using NewUpdateKV to apply transactional updates under consistency models with $\texttt{vShift}_{\texttt{MR}\cap\texttt{RYW}}$ saves us the `vShift` check without changing the set of reachable kv-stores compared with the original transactional update definition.

6.2 Other Models

The view-shift condition for the Update Atomic (`UA`) consistency model is defined by $\mathtt{vShift}_{\mathtt{UA}} \overset{def}{=} \mathtt{true}$. That is, a client may lose information about the kv-store after committing a transaction. Since NewUpdateKV satisfies $\mathtt{vShift}_{\mathtt{MR}\cap\mathtt{RYW}}$, it is clear that the views allowed, and therefore kv-stores reachable, are affected by NewUpdateKV.

The `UA` model is not considered a useful model. It was introduced for the purpose of defining `SI` compositionally from `UA` and `CP` (consistent prefix). A variation of `UA` using the $\mathtt{vShift}_{\mathtt{MR}\cap\mathtt{RYW}}$ condition is suggested in [1], which can be improved by using our new transactional update definition.

Chapter 7

Implementation

This project started as an implementation project, with the initial goal to implement the semantics of Xiong et al. [10], and investigate ways to use that implementation for automatic litmus test generation.

An implementation of the semantics involves simulating the execution of client programs on a kv-store and identifying candidate programs that capture differences in behaviour under different consistency models. This might be done by examining all final kv-store states obtained by executing a program under different consistency models and spotting states that are only reachable under one of the models.

Throughout the development process, our most fundamental goal has been to create an implementation that is not only computationally efficient, but also elegant in the way it exploits properties of the semantics and specific consistency models.

In section 7.1 we give an overview of our original plan. Section 7.2 explains the first steps we took towards implementing the CAtomicTrans rule. We decided to focus on the causal consistency `CC` model because a view that satisfies the `CC` closure property remains in `CC` after a transactional update. We explain how our implementation works under `CC` in section 7.3.

7.1 Overview

Our vision for the final product is a program that, given an input client program, generates all reachable kv-stores under two given consistency models, compares them to find the execution paths that result in a state that is reachable under one model but not the other, and returns that information. This information would be analysed to construct a litmus test.

Because of similarities in structure, and familiarity of the research group, OCaml was the natural language choice for the group and the research community.

The first step is modelling components of the semantics using appropriate data structures. We explain our choices for the main components: kv-stores, fingerprints, and views.

We choose to model a kv-store using a mutable hash table from keys (integers) to lists of versions, where a list is stored in reverse order to allow fast access to newer versions. Each version is represented by a record carrying a value, a transaction (record), and a mutable and initially empty set of transactions representing the reader set. This allows updates to a kv-store to be reflected globally.

A fingerprint is a set of tuples, each with three elements corresponding to $(\mathtt{W}, k, v)$: the type of operation (`W` or `R`), the key, and the value.

Views are hash tables from keys to sets of integers representing indices of versions in the kv-store.

7.2 Implementing CAtomicTrans

Starting to implement the CAtomicTrans rule, we decided to focus first on applying a single transactional update given a kv-store $\mathcal{K}$ and a transaction t with fingerprint $\mathcal{F}$, skipping the first three premises.

We provide an implementation of the original UpdateKV function (definition 3.12), as well as the NewUpdateKV function which updates the write-view (definition 4.4). Both functions use a transaction identifier t obtained from our implementation of the NextTxID function (definition 3.11) that returns the next available transaction identifier for a client cl given a kv-store $\mathcal{K}$.

As we developed our understanding of the semantics we adapted our implementation to support our ideas. This is demonstrated by our implementation of the NewUpdateKV and NextTxID functions.

It became clear that while a kv-store will be shared between multiple clients, the various possibilities for interleaving transactions and using different initial views for the same transaction mean that multiple versions of the kv-store must be stored and shared between clients.

To allow saving histories and using different views to obtain all possible kv-stores reachable by executing a given program, we create a copy of the kv-store before applying an update. Similarly, when NewUpdateKV is called we create a copy of the client view to represent the write-view u_w to which we add the new versions. This is another instance where the implementation has been affected by our evolving understanding.

7.3 Causal Consistency (CC)

Our implementation allows a full step by step run of a program under CC.

As explained in section 4.3, the implementation of NewUpdateKV incorporates the minimal view shift that satisfies $\texttt{vShift}_{\texttt{MR}\cup\texttt{RYW}}$. Since the CC closure relation is defined by $R_{\texttt{CC}} \overset{def}{=} \mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}$, if we choose not to advance client views at the beginning of a transaction (section 3.2) we would always read our writes, and the $\mathsf{WR}_{\mathcal{K}}$ relation would be a subset of the SO relation.

Next we considered generating fingerprints from programs, and computing the closure of the `canCommit` predicate. We provide an implementation of the combination operator (explained in section 3.2) to generate a well-formed fingerprint given operations.

An interesting property of causal consistency is that its `canCommit` check does not depend on the fingerprint of the executing transaction. This opens up possibilities for advancing the client view when starting a transaction in a forward way that reduces the computation of, and possibly eliminates, the `canCommit` check in a way similar to our NewUpdateKV implementation eliminating the need for `vShift`.

Recognising that a naive implementation would be very cumbersome and computationally complex, we agreed that advancing the theory first by exploiting this property, and other properties of the semantics and different consistency models, would be better both from a research contribution perspective and for providing the foundations for an efficient, and elegant, future implementation.

Chapter 8

Exploring Causal Consistency (CC)

We choose to focus on the causal consistency model. Causal consistency states that, if a version written by transaction t is included in the view of a client before committing a transaction, then all versions that t observes must also be in the client view [10]. This model is unique in that it captures only the dependencies associated with the executing client, and does not depend on the specific transaction being executed. This is reflected in the $\texttt{canCommit}_{\texttt{CC}}$ predicate (fig. 3.2), which does not depend on the fingerprint $\mathcal{F}$ but only on the kv-store and the client's view over the store. In particular, given a client view in CC (that satisfies the CC execution test), then the minimal view resulting from an update of a transaction is also in CC. Recall that the closure relation for CC is $R_{\texttt{CC}} \overset{def}{=} \mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}$. We can therefore explore CC properties and think about $\texttt{closed}(\mathcal{K}, u_w, R_{\texttt{CC}})$ (definition 3.10) without associating it with a particular transactional update.

Observe that, as with other properties (section 5.2), showing that a CC view remains closed after a transactional update (section 8.1), between transactions (section 8.2), and when advancing the view at the start of a transaction to incorporate environment steps not done by the client (section 8.3), would allow us to eliminate the need for the `canCommit` check under causal consistency.

We identify that understanding how to incorporate environment information into the client view when starting a transaction is key to developing an elegant implementation. Using our notion of *minimal views* (section 5.1), we devise a systematic way of extending client views to incorporate information from the environment in section 8.3. We extend our approach to snapshots and provide an algorithm for obtaining all possible kv-stores using an altered definition of a snapshot.

8.1 Transactional Update and CC

We show that when applying a transactional update using NewUpdateKV on a view that is closed with respect to a kv-store and R_{CC}, the resulting view is closed with respect to the updated kv-store and R_{CC}.

Proposition 8.1 (NewUpdateKV preserves CC)**.** *Given a kv-store $\mathcal{K} \in$ KVS, views $u_r, u_w \in \textsc{Views}(\mathcal{K})$, a fingerprint $\mathcal{F} \subseteq \textsc{Op}$, and a transaction identifier $t \in \textsc{TxID}_0$,*

$$\mathtt{closed}(\mathcal{K}, u_w, R_{\mathsf{CC}}) \wedge (\mathcal{K}', u') = \mathsf{NewUpdateKV}(\mathcal{K}, u_r, u_w, \mathcal{F}, t) \Rightarrow \mathtt{closed}(\mathcal{K}', u', R_{\mathsf{CC}})$$

where $\mathtt{closed}(\mathcal{K}, u, R)$ *is given in definition 3.10.*

Proof sketch. Intuitively, the CC closure is about observations of the client session order and its reads. Since the initial view is in CC, all previous transactions by cl and their reads are in the view. If the executing transaction reads a version, that version must have been in the initial view (snapshot property) and all transactions R^*_{CC}-before its writer are in the visibility set. Since we add our writes to the write view, SO is also maintained. □

Proof. We prove proposition 8.1 by induction on the size of the fingerprint $\mathcal{F}$.

1. **Base Case** $|\mathcal{F}| = 0$.

   In this case $\mathcal{F} = \emptyset$. proposition 8.1 trivially holds as $u' = u_w$ and $\mathcal{K}' = \mathcal{K}$ by definition 4.4.

2. **Inductive Case** $|\mathcal{F}| > 0$.

   Now $\mathcal{F} = \{o\} \uplus \mathcal{F}'$ for some operation $o \in \textsc{Op}$ and fingerprint $\mathcal{F}' \in \textsc{Fp}$ where o is either a read or a write.

   (a) **Case** $\mathcal{F} = \{(\mathtt{R}, k, v)\} \uplus \mathcal{F}'$. (read operation)
   Let index $i = \max_<(u_r(k))$ and version $(v, t', T) = \mathcal{K}(k, i)$. Let $\mathcal{K}'' = \mathcal{K}[k \mapsto \mathcal{K}(k)[i \mapsto (v, t', T \cup \{t\})]]$. There is no change to $\mathsf{visTx}(\mathcal{K}, u_w)$ as u_w is unchanged, $\mathsf{visTx}(\mathcal{K}, u_w) = \mathsf{visTx}(\mathcal{K}'', u_w)$.
   We have that $(t', t) \in \mathsf{WR}_{\mathcal{K}''}(k)$. Since t' is in $\mathsf{visTx}(\mathcal{K}, u_w)$, all transactions that are R^*_{CC}-before t and are not read-only are in $\mathsf{visTx}(\mathcal{K}, u_w)$.
   We get that the closure

   $$\begin{aligned}&\mathsf{visTx}(\mathcal{K}, u_w) = \\ &\mathsf{visTx}(\mathcal{K}'', u_w) = \\ &\left\{t' \mid t \in \mathsf{visTx}(\mathcal{K}'', u_w) \wedge t' \in (R^*_{\mathsf{CC}})^{-1}(t)\right\} \setminus \{t \mid \forall k, i.\, t \neq \mathsf{w}(\mathcal{K}''(k, i))\}\end{aligned}$$

   as required.

(b) **Case** $\mathcal{F} = \{(\mathtt{W}, k, v)\} \uplus \mathcal{F}'$. (write operation)
Let $\mathcal{K}'' = \mathcal{K}\left[k \mapsto \mathcal{K}(k) :: (v, t, \emptyset)\right]$ and $u'_w = u_w\left[k \mapsto u_w(k) \uplus \{|\mathcal{K}(k)|\}\right]$. As we are looking at the reflexive-transitive closure of R_{CC}, it is clear to see that new transactions do not affect previous closures. Therefore, we only need to look at the new transaction. By adding the new version to u'_w the executing transaction t is included in $\mathsf{visTx}(\mathcal{K}'', u'_w)$. The only transaction added to the set of R^*_{CC}-before t is t itself. We get that

$$\begin{aligned}&\mathsf{visTx}(\mathcal{K}'', u'_w) = \\ &\left\{t' \mid t \in \mathsf{visTx}(\mathcal{K}'', u'_w) \wedge t' \in (R^*_{\mathsf{CC}})^{-1}(t)\right\} \setminus \{t \mid \forall k, i.\, t \neq \mathsf{w}(\mathcal{K}''(k, i))\}\end{aligned}$$

as required. □

8.2 Between Transactions

We show that updates to the kv-store by other clients do not affect the closure of a view u of client cl.

Proposition 8.2 (Later kv-store and CC). *Given kv-stores $\mathcal{K}, \mathcal{K}' \in \mathtt{KVS}$, and a view $u \in \textsc{Views}(\mathcal{K})$ for a client cl,*

$$\mathtt{closed}(\mathcal{K}, u, R_{\mathsf{CC}}) \wedge \mathcal{K} \sqsubseteq \mathcal{K}' \wedge \forall n.\, t^n_{cl} \notin \mathcal{K}' \setminus \mathcal{K} \Rightarrow \mathtt{closed}(\mathcal{K}', u, R_{\mathsf{CC}}).$$

Proof sketch. Intuitively, once a view u is closed with respect to a kv-store $\mathcal{K}$ and R_{CC} future transactions cannot affect the session order before visible transactions in u. Similarly, once a transaction has been committed it cannot read new versions, so future transactions do not affect the write-read dependencies of visible transactions in u. □

8.3 Environment Information and CC

We have proved that applying a transactional update using NewUpdateKV on a kv-store $\mathcal{K}$ and view u in CC results in the updated kv-store $\mathcal{K}'$ and view u' satisfying $\mathtt{closed}(\mathcal{K}', u', R_{\mathsf{CC}})$, where u' is the minimal view after the transaction.

We have also shown that the view u over a kv-store $\mathcal{K}$ of client cl remains in CC when other clients update the kv-store between cl's transactions.

We show that our minimal views are sufficient to progress client views to reflect the effect of the environment on the kv-store in a forward way. We consider multiple ways for achieving this.

A naive way for extending the client view u at the start of a transaction would be to generate all possible views u''s that are $u \sqsubseteq u''$, and discarding views that are not in CC by computing their closure. In the same way that we incorporated $\texttt{vShift}_{\texttt{MR}\cap\texttt{RYW}}$ into our transactional update, we are interested in understanding what it means to find such u''s cleverly in a forward way.

We explore different ways to advance the view of a client. We look at the question from two separate points of view: implementation and client.

- Implementation knowledge: it is reasonable to assume that the implementation has complete knowledge of the kv-store and all client views and their histories. This means that from this point of view, we are able to determine the views and snapshots of any transaction.
- Client knowledge: a client has knowledge of its current view on the kv-store and access to the kv-store.

By looking at the question from these perspectives we aim to find a balance between the amount of information needed to be stored and the complexity of computation.

We start by looking at combining views from different clients to advance a view.

Definition 8.3 (Union of views). *Given two views $u, u' \in \textsc{Views}(\mathcal{K})$, the union of the views is defined by*

$$(u \sqcup u')(k) = u(k) \cup u'(k)$$

for every $k \in \mathrm{dom}(\mathcal{K})$.

By taking the union of two views we are able to create a new view that is more advanced than either view. We prove that the union of two CC views $u, u' \in \textsc{Views}(\mathcal{K})$ on a kv-store $\mathcal{K}$ is still in CC.

Proposition 8.4 (Union of CC views). *Given a kv-store $\mathcal{K}$ and views $u, u' \in \textsc{Views}(\mathcal{K})$,*

$$\texttt{closed}(\mathcal{K}, u, R_{\mathsf{CC}}) \wedge \texttt{closed}(\mathcal{K}, u', R_{\mathsf{CC}}) \Rightarrow \texttt{closed}(\mathcal{K}, u \sqcup u', R_{\mathsf{CC}}).$$

Proof sketch. Intuitively, since both views are closed we need to show that relations between transactions from $\mathsf{visTx}(\mathcal{K}, u)$ and $\mathsf{visTx}(\mathcal{K}, u')$ do not expand the closure (clearly, the union of the closures is $\subseteq$ the closure of the union). Take $t \in \mathsf{visTx}(\mathcal{K}, u)$, then all the transactions t' that are R^*_{CC}-before t, i.e. $t' \in (R^*)^{-1}(t)$, and are not read-only transactions $t' \notin \{t'' \mid \forall k, i.\, t'' \neq \mathsf{w}(\mathcal{K}(k, i))\}$, are also visible in u, i.e. $t' \in \mathsf{visTx}(\mathcal{K}, u)$. Similarly for $t \in \mathsf{visTx}(\mathcal{K}, u')$. □

Proof. We want to show

$$\mathsf{visTx}(\mathcal{K}, u \sqcup u') = \left((R^*_{\mathsf{CC}})^{-1}(\mathsf{visTx}(\mathcal{K}, u \sqcup u'))\right) \setminus \{t \mid \forall k, i.\, t \neq \mathsf{w}(\mathcal{K}(k, i))\}$$

We have that

$$\mathsf{visTx}(\mathcal{K}, u) = \left((R^*_{\mathsf{CC}})^{-1}(\mathsf{visTx}(\mathcal{K}, u))\right) \setminus \{t \mid \forall k, i.\, t \neq \mathsf{w}(\mathcal{K}(k, i))\} \tag{8.1}$$
$$\mathsf{visTx}(\mathcal{K}, u') = \left((R^*_{\mathsf{CC}})^{-1}(\mathsf{visTx}(\mathcal{K}, u'))\right) \setminus \{t \mid \forall k, i.\, t \neq \mathsf{w}(\mathcal{K}(k, i))\} \tag{8.2}$$

If we union eq. (8.1) and eq. (8.2) we get

$$\begin{aligned}&\mathsf{visTx}(\mathcal{K}, u) \cup \mathsf{visTx}(\mathcal{K}, u') = \\ &\qquad\left(\left((R^*_{\mathsf{CC}})^{-1}(\mathsf{visTx}(\mathcal{K}, u))\right) \setminus \{t \mid \forall k, i.\, t \neq \mathsf{w}(\mathcal{K}(k, i))\}\right) \cup \\ &\qquad\left(\left((R^*_{\mathsf{CC}})^{-1}(\mathsf{visTx}(\mathcal{K}, u'))\right) \setminus \{t \mid \forall k, i.\, t \neq \mathsf{w}(\mathcal{K}(k, i))\}\right)\end{aligned}$$

From definition (ref visTx def), it is clear to see that $\mathsf{visTx}(\mathcal{K}, u \sqcup u') = \mathsf{visTx}(\mathcal{K}, u) \cup \mathsf{visTx}(\mathcal{K}, u')$. So we get

$$\begin{aligned}&\mathsf{visTx}(\mathcal{K}, u \sqcup u') = \\ &\left(\left((R^*_{\mathsf{CC}})^{-1}(\mathsf{visTx}(\mathcal{K}, u))\right) \cup \left((R^*_{\mathsf{CC}})^{-1}(\mathsf{visTx}(\mathcal{K}, u'))\right)\right) \setminus \{t \mid \forall k, i.\, t \neq \mathsf{w}(\mathcal{K}(k, i))\}\end{aligned}$$

All that remains to be shown is

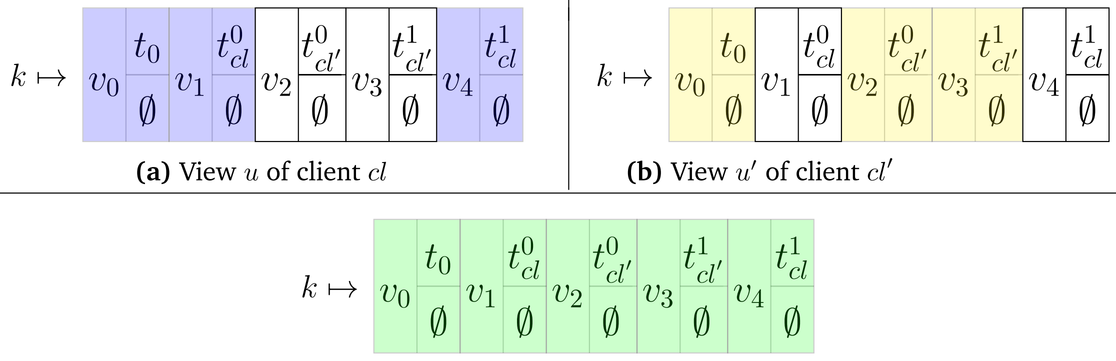
$$\begin{aligned}&\left((R^*_{\mathsf{CC}})^{-1}(\mathsf{visTx}(\mathcal{K}, u \sqcup u'))\right) \setminus \{t \mid \forall k, i.\, t \neq \mathsf{w}(\mathcal{K}(k, i))\} = \\ &\quad\left(\left((R^*_{\mathsf{CC}})^{-1}(\mathsf{visTx}(\mathcal{K}, u))\right) \cup \left((R^*_{\mathsf{CC}})^{-1}(\mathsf{visTx}(\mathcal{K}, u'))\right)\right) \setminus \{t \mid \forall k, i.\, t \neq \mathsf{w}(\mathcal{K}(k, i))\}\end{aligned}$$

Unfolding the definitions

$$\begin{aligned}&\left((R^*_{\mathsf{CC}})^{-1}(\mathsf{visTx}(\mathcal{K}, u \sqcup u'))\right) \setminus \{t \mid \forall k, i.\, t \neq \mathsf{w}(\mathcal{K}(k, i))\} = \\ &\left\{t' \mid t \in \mathsf{visTx}(\mathcal{K}, u \sqcup u') \land t' \in (R^*_{\mathsf{CC}})^{-1}(t)\right\} \setminus \{t \mid \forall k, i.\, t \neq \mathsf{w}(\mathcal{K}(k, i))\} = \\ &\left\{t' \mid t \in (\mathsf{visTx}(\mathcal{K}, u) \cup \mathsf{visTx}(\mathcal{K}, u')) \land t' \in (R^*_{\mathsf{CC}})^{-1}(t)\right\} \setminus \{t \mid \forall k, i.\, t \neq \mathsf{w}(\mathcal{K}(k, i))\} = \\ &\left\{t' \mid (t \in \mathsf{visTx}(\mathcal{K}, u) \lor t \in \mathsf{visTx}(\mathcal{K}, u')) \land t' \in (R^*_{\mathsf{CC}})^{-1}(t)\right\} \setminus \{t \mid \forall k, i.\, t \neq \mathsf{w}(\mathcal{K}(k, i))\} = \\ &\left\{t' \mid \left(t \in \mathsf{visTx}(\mathcal{K}, u) \land t' \in (R^*_{\mathsf{CC}})^{-1}(t)\right) \lor \left(t \in \mathsf{visTx}(\mathcal{K}, u') \land t' \in (R^*_{\mathsf{CC}})^{-1}(t)\right)\right\} \\ &\qquad \setminus \{t \mid \forall k, i.\, t \neq \mathsf{w}(\mathcal{K}(k, i))\} = \\ &\left\{t' \mid t \in \mathsf{visTx}(\mathcal{K}, u) \land t' \in (R^*_{\mathsf{CC}})^{-1}(t)\right\} \cup \left\{t' \mid t \in \mathsf{visTx}(\mathcal{K}, u') \land t' \in (R^*_{\mathsf{CC}})^{-1}(t)\right\} \\ &\qquad \setminus \{t \mid \forall k, i.\, t \neq \mathsf{w}(\mathcal{K}(k, i))\} = \\ &\left(\left((R^*_{\mathsf{CC}})^{-1}(\mathsf{visTx}(\mathcal{K}, u))\right) \cup \left((R^*_{\mathsf{CC}})^{-1}(\mathsf{visTx}(\mathcal{K}, u'))\right)\right) \setminus \{t \mid \forall k, i.\, t \neq \mathsf{w}(\mathcal{K}(k, i))\}\end{aligned}$$

as required. □

Consider the implementation perspective where, for every transaction identifier t in $\mathcal{K}$, we record the transaction pair (t, u) where u is the immediate view after t has been committed, which we know is in CC (proposition 8.1). The view u is the minimal view with respect to the update (definition 5.1).


$k \mapsto$
v_0 t_0 $\emptyset$
v_1 t_{cl}^0 $\emptyset$
v_2 $t_{cl'}^0$ $\emptyset$
v_3 $t_{cl'}^1$ $\emptyset$
v_4 t_{cl}^1 $\emptyset$

(a) View u of client cl

(b) View u' of client cl'

(c) The view $u'' = u \sqcup u'$

Figure 8.1: Union of CC views where neither subsumes the other resulting in the same snapshot for client cl

Now consider client cl which previously committed t with the update view u: that is, the pair (t, u) in CC. The problem is to find the possible u'' in a forward way, without having to compute the closure.

We use $i_1, i_2, \dots \in \{1, \dots, n\}$ to represent clients $cl_1, cl_2, \dots, cl_n$ in a kv-store. We use $j_1, j_2, \dots \in \{1, \dots, m\}$ to represent the transaction identifier number e.g. t_{i1}^{j2} means the j_2th transaction of client cl_{i1}.

First approach Let client cl have a view u over the kv-store $\mathcal{K}$. Consider all the (t_{i1}^{j1}, u_{i1j1}) pairs in the store for clients cli_1 with $i_1 \in \{1, \dots, n\}$ and $j_1 \in \{1, \dots, m\}$. Now form $(t\, t_{i1}^{j1}, u \sqcup u_{i1j1})$ where $u \sqcup u_{i1j1}$ in CC is the union of views and $t\, t_{i1}^{j1}$ is some identifier used to represent the transactions used for all combinations of i_1 and j_1 in the kv-store. Now form $(t\, t_{i1}^{j1}\, t_{i2}^{j2}, (u \sqcup u_{i1j1}) \sqcup u_{i2j2})$ for $i_2 \in \{1, \dots, n\}$ and $j_2 \in \{1, \dots, m\}$ and $(u \sqcup u_{i1j1}) \sqcup u_{i2j2}$ in CC. Keep going until no new views are created.

Have we found all possible u''? Since we are working with *minimal views* (section 5.1), we are able to find all possible u''s using this approach. Moreover, this approach is forward in the way that all resulting views are known to be in CC. However, it is clear that this is not an ideal approach.

We observe that the union of some views is more interesting than others. As a simple example, if we union the view u of client cl with a view u' that is ordered before $u' \sqsubseteq u$ then the resulting view $u = u \sqcup u'$ is the same. Furthermore, advancing a view u to a view u' that is strictly greater than u does not necessarily mean that the *snapshot* of the transaction would change. This is demonstrated in fig. 8.1. If the snapshot is the same, then the effect of the transaction on the kv-store would be the same, because read operations access the same versions. We are therefore interested in finding *meaningful views*, meaning views that result in a different snapshot.

For this reason, we do not need to work with all transaction identifiers in the store. We consider only transaction identifiers to the right of the snapshot of cl's view. That is, writers of versions committed to the kv-store after the newest version client cl observes in its view.

Second approach (better) Let client cl have a view u over the kv-store $\mathcal{K}$. Now consider all the (t_{i1}^{j1}, u_{i1j1}) pairs in the store *to the right* of the snapshot of u on $\mathcal{K}$, $\sigma = \mathsf{snapshot}(\mathcal{K}, u)$, for clients cli_1 with $i1 \in \{1, \ldots, n\}$ and $j_1 \in \{1, \ldots, m\}$. As before, form $(t\, t_{i1}^{j1}, u \sqcup u_{i1j1})$ where $u \sqcup u_{i1j1}$ in CC is the union of views and $t\, t_{i1}^{j1}$ is some identifier used to represent the transactions used for all combinations of i_1 and j_1 in the kv-store to the right of σ. Now form $(t\, t_{i1}^{j1}\, t_{i2}^{j2}, (u \sqcup u_{i1j1}) \sqcup u_{i2j2})$ for $i_2 \in \{1, \ldots, n\}$ and $j_2 \in \{1, \ldots, m\}$ and $(u \sqcup u_{i1j1}) \sqcup u_{i2j2}$ in CC. Continue until no new views are created.

Have we found all meaningful u''? We know that the union of client cl's view u with views of transactions to the right of the snapshot σ will generate a view with a new snapshot. Using this approach we are able to find all meaningful u''s. As with our previous approach, all resulting views are known to be in CC.

We notice that what actually matters for analysing kv-stores is the snapshot, not the view. This is because the fingerprint, and therefore the effect, of a transaction on the kv-store is determined by its snapshot (section 3.2). We consider using our approach on snapshots rather than views.

Definition 8.5 (Max of snapshots). *Given snapshots* $\sigma = \mathsf{snapshot}(\mathcal{K}, u)$ *and* $\sigma' = \mathsf{snapshot}(\mathcal{K}, u')$ *for a kv-store* $\mathcal{K}$ *and views* $u, u' \in \textsc{Views}(\mathcal{K})$*, the* max *of the snapshots is defined by the snapshot of the union* $u \sqcup u'$ *in* $\mathcal{K}$*,* $\mathsf{snapshot}(\mathcal{K}, u \sqcup u')$

where $\mathsf{snapshot}(\mathcal{K}, u)$ *is defined in definition 3.7.*

But this still requires storing the full view.

Recall that a snapshot is a function from keys to the most up-to-date values a client can observe in the kv-store, and that the fingerprint is obtained from the snapshot. When we obtain the snapshot of the union of two views we effectively update the most up-to-date value of each key to the newest value that either of the views included. But if we know that there exists a view in CC that results in some snapshot, then knowing the whole view does not change the effect of a transaction. We alter the definition of a snapshot to take advantage of this property.

We define a new notion of a snapshot, version-snapshot, that is a function from key k to value v as before, but also to the index of the version whose value is v.

Definition 8.6 (Version-snapshots). *Given* $\mathcal{K} \in \mathrm{KVS}$ *and* $u \in \textsc{Views}(\mathcal{K})$*, the* version-snapshot *of* u *in* $\mathcal{K}$ *is a function,* $\mathsf{verSnapshot}(\mathcal{K}, u) : \textsc{Key} \to \textsc{Value} \times \mathbb{N}$*, that is defined by:*

$$\mathsf{verSnapshot}(\mathcal{K}, u)(k) \overset{def}{=} (i, \mathsf{val}(\mathcal{K}(k, i)))$$

for all $k \in \textsc{Key}$, where $i = \max_{<}(u\,(k))$, the maximum element in $u\,(k)$ w.r.t. the natural order $<$ over $\mathbb{N}$.

Definition 8.7 (Max of version-snapshots). *Given version-snapshots σ_v and σ'_v, the* max *of the version-snapshots is defined by:*

$$\max(\sigma_v, \sigma'_v)(k) \overset{def}{=} \max_{\leq}(\sigma_v(k), \sigma'_v(k))$$

for all $k \in \textsc{Key}$.

Notice that we use $\max_{\leq}$ because the index-value pair for some keys might be the same in both version-snapshots.

Let the initial version-snapshot for all clients $\sigma_{v_0} = \mathsf{verSnapshot}\,(\mathcal{K}_0, u_0)$ where $\mathcal{K}_0$ is the initial kv-store and u_0 is the initial view on $\mathcal{K}_0$.

For each version-snapshot σ_v, we know that there exists some view u on the kv-store $\mathcal{K}$ such that $\sigma_v = \mathsf{verSnapshot}\,(\mathcal{K}, u)$. We can now consider a third approach, maxing version snapshots.

Third approach (better) Consider all the (t, σ_v) pairs in the store where σ_v is the version-snapshot of u in the transaction pair (t, u). We follow the same approach as before, but instead of taking all possible unions of views of transactions to the right of the snapshot of client cl's view, we max the version snapshot σ_v with all possible version snapshots to the right in the kv-store. We continue until no new version-snapshots are created.

Have we found all snapshots of meaningful u''? Since we know the existence of a view on the kv-store for each version-snapshot, we are able to find all version-snapshots of meaningful u'' with this approach. As with our previous approach, all resulting version-snapshots are known to have views in CC.

Taking this a step further, we can redefine our transactional update to work on version-snapshots, rather than views, with a read-snapshot for read operations and a minimal version-snapshot constructed by updating the index-value pair for key k for each $(\mathtt{W}, k, v) \in \mathcal{F}$ to be $(|\mathcal{K}\,(k)|\,, v)$, similarly to our write-view.

Definition 8.8 (Transactional update on version-snapshots). *Given a kv-store $\mathcal{K} \in$ KVS, version-snapshots σ_{vr}, σ_{vw}, a fingerprint $\mathcal{F}$, and a transaction identifier $t \in \textsc{TxID}_0$, the transactional update function on version-snapshots* $\mathsf{VSnUpdateKV}\,(\mathcal{K}, \sigma_{vr}, \sigma_{vw}, \mathcal{F}, t)$ *is defined by:*

$$\mathsf{VSnUpdateKV}\,(\mathcal{K}, \sigma_{vr}, \sigma_{vw}, \emptyset, t) \overset{def}{=} (\mathcal{K}, \sigma_{vw})$$

$$
\begin{aligned}
&\mathsf{VSnUpdateKV}\left(\mathcal{K}, \sigma_{vr}, \sigma_{vw}, \{(\mathtt{R}, k, v)\} \uplus \mathcal{F}, t\right) \overset{def}{=} \\
&\qquad \mathtt{let}\ (i, v) = \sigma_{vr}(k)\ \mathtt{and}\ (v, t', T) = \mathcal{K}\left(k, i\right)\ \mathtt{in} \\
&\qquad\quad \mathsf{VSnUpdateKV}\left(\mathcal{K}\left[k \mapsto \mathcal{K}\left(k\right)\left[i \mapsto (v, t', T \uplus \{t\})\right]\right], \sigma_{vr}, \sigma_{vw}, \mathcal{F}, t\right) \\
&\mathsf{VSnUpdateKV}\left(\mathcal{K}, \sigma_{vr}, \sigma_{vw}, \{(\mathtt{W}, k, v)\} \uplus \mathcal{F}, t\right) \overset{def}{=} \\
&\qquad \mathtt{let}\ \mathcal{K}' = \mathcal{K}\left[k \mapsto \mathcal{K}\left(k\right) :: (v, t, \emptyset)\right]\ \mathtt{and}\ \sigma'_{vw} = \sigma_{vw}\left[k \mapsto \left(\left|\mathcal{K}\left(k\right)\right|, v\right)\right]\ \mathtt{in} \\
&\qquad\qquad \mathsf{VSnUpdateKV}\left(\mathcal{K}', \sigma_{vr}, \sigma'_{vw}, \mathcal{F}, t\right)
\end{aligned}
$$

Intuitively, each version-snapshot has exactly one corresponding snapshot. There may be more than one view resulting in that snapshot, but it is enough to know the existence of a view on the kv-store with the newest version for each key corresponding to the version-snapshot as the effect of the transaction would be the same.

We give an overview of a possible algorithm for implementing our approach. We start with the initial kv-store $\mathcal{K}_0$ and initial version-snapshot for each client σ_{v_0}. Imagine that this is the root of a tree whose branches represent possible execution paths. Starting from the root, we apply all possible first transactions given the input client program. For each transaction, we create a new child node with the resulting kv-store after update, and carry the version snapshots for all clients from the previous node. We max each version-snapshot with the minimal version-snapshot of the committed transaction, as well as keep the previous version snapshot. This results in one version-snapshot for the committed transaction, and two possible version snapshots for each of the other clients. Similarly to our first step, we apply all possible next transactions at each node. If the next transaction we apply is committed by some client cl, then we compute its fingerprint on each possible version-snapshot of that client, thus creating a new child node for each possible update. As before, the committed transaction has one version-snapshot in each child node, the minimal version-snapshot of its update. We carry the previous version-snapshots and max with the resulting one to create all possible version-snapshots for the latest transaction by each client. We continue in this way until all transactions have been committed, resulting in all kv-stores and possible version-snapshots on the store.

To simulate a single execution path, follow the same approach but only compute the fingerprint for a transaction by client cl on one of its possible version-snapshots in the node, ignoring the rest.

In chapter 9 we suggest looking into optimising this strategy by analysing the keys with which a transaction interacts to avoid repeating branches. Specific implementation optimisations can be considered, for example 'forgetting' about old versions of a key in the kv-store after all version-snapshots of all clients in the node have newer versions of that key in their version-snapshot.

In our suggested approach, the implementation should keep the minimal version-snapshot after each update. This seems to be quite reasonable considering the potential performance gain. We suspect that it might be possible to store even less information by analysing the fingerprints of transactions and only considering subsets of the keys that the client observes. However, we believe this is more relevant in models whose closure relation depends on the fingerprint.

Chapter 9

Conclusions and Future Work

On balance, this project was a resounding success. This report demonstrates that it clearly fulfils our aims of reformulating the semantics of Xiong et al. in a forward-looking way, and went further to explain the behaviour of transactions under causal consistency as well as provide an algorithm for directly incorporating environment information to obtain all reachable kv-stores.

Xiong et al. [10] introduced an interleaving centralised operational semantics for capturing the client-observable behaviour of distributed atomic transactions. We evaluate their semantics from an implementation perspective and recognise that the backwards analysis required for transactional updates is too complicated. We give a forward-looking reformulation, following the spirit of more traditional operational semantics, and prove that it is equivalent to the original. We provide and explore an implementation of some of the main components of the semantics. Realising that a key step to developing an elegant implementation is understanding how to incorporate environment steps not done by the client into its view when starting a transaction, we choose to investigate specific consistency models.

We focus on the causal consistency (CC) model, and propose two different perspectives for looking at the view shift question. We identify the notion of minimal views and use it to demonstrate a systematic way to incorporate the environment information to obtain all the possible extended CC views for a client. We modify the definition of a snapshot and provide a more efficient way to directly obtain all the snapshots of *meaningful views* of a client executing a transaction. We provide an algorithm for implementing our approach. We believe that our approach provides a good balance between computational efficiency and space complexity. We propose future extensions to this project.

As discussed in chapter 8, causal consistency is unique in that transactional updates result in minimal CC views, as the closure relation, $R_{\mathtt{CC}}$, does not depend on the fingerprint. This is not the case with other models such as PSI and SI. In the same way that we explored causal consistency, future work may wish to investigate other consistency models to design efficient approaches for their implementations.

Future work may wish to implement our approach (section 8.3) and `CC` algorithm (section 8.3). By implementing another model, either serialisability (`SER`) or a weaker model such as parallel snapshot isolation (`PSI`), research focus can shift back to automatic litmus test generation. As a first step, by reproducing known anomalies such as the lost update anomaly (example 2.1) disallowed by `PSI` and `SI` but allowed under `CC`, and the long fork anomaly (example 2.2) disallowed under `SER` and `SI` but allowed under `PSI` and `CC`. Then, by looking into recognising 'interesting' candidate execution paths and focusing on them instead of producing all obtainable kv-stores.

We believe that further optimisations to our version-snapshots approach (section 8.3) can be made by looking at subsets of the snapshot, based on the fingerprint. From the implementation perspective, we suggest investigating the effect of maxing the client version-snapshot with the version-snapshots of transactions to the right of the client snapshot in the kv-store only for the keys with which the transaction interacts (reads from or writes to). Further, implementation optimisations to our algorithm can be considered, for example 'forgetting' about old versions of a key in the kv-store after all version-snapshots of all clients in the node have newer versions of that key in their version-snapshots (section 8.3). From the client perspective, we suggest looking into working out the *partial* minimal snapshots of transactions in the kv-store by examining their reads and writes to reduce the amount of computation needed to compute the closure. This applies to `CC` but is even more relevant for models that depend on the fingerprint e.g. `PSI` and `SI`.

Bibliography

[1] Xiong S. Parametric Operational Semantics for Consistency Models. Imperial College London; 2021. Available from: `http://www.shalexiong.com/thesis.pdf`. pages 1, 6, 7, 10, 14, 18, 32, 41

[2] Ardekani MS, Sutra P, Shapiro M. Non-Monotonic Snapshot Isolation: Scalable and Strong Consistency for Geo-Replicated Transactional Systems. In: Proceedings of the 2013 IEEE 32nd International Symposium on Reliable Distributed Systems. SRDS '13. USA: IEEE Computer Society; 2013. p. 163–172. Available from: `https://doi.org/10.1109/SRDS.2013.25`. pages 1, 7

[3] Bailis P, Fekete A, Ghodsi A, Hellerstein JM, Stoica I. Scalable Atomic Visibility with RAMP Transactions. ACM Trans Database Syst. 2016 July;41(3). Available from: `https://doi.org/10.1145/2909870`. pages 1

[4] Berenson H, Bernstein P, Gray J, Melton J, O'Neil E, O'Neil P. A Critique of ANSI SQL Isolation Levels. In: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data. SIGMOD'95. ACM; 1995. p. 1–10. pages 1, 6, 10

[5] Binnig C, Hildenbrand S, Färber F, Kossmann D, Lee J, May N. Distributed Snapshot Isolation: Global Transactions Pay Globally, Local Transactions Pay Locally. The VLDB Journal. 2014 December;23(6):987–1011. pages 1

[6] Du J, Elnikety S, Zwaenepoel W. Clock-SI: Snapshot Isolation for Partitioned Data Stores Using Loosely Synchronized Clocks. In: Proceedings of the 32nd Leibniz International Proceedings in Informatics (LIPIcs). SRDS'13. Washington, DC, USA: IEEE Computer Society; 2013. p. 173–184. Available from: `https://doi.org/10.1109/SRDS.2013.26`. pages 1

[7] Lloyd W, Freedman MJ, Kaminsky M, Andersen DG. 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. SOSP '11. New York, NY, USA: Association for Computing Machinery; 2011. p. 401–416. Available from: `https://doi.org/10.1145/2043556.2043593`. pages 1

[8] Raad A, Lahav O, Vafeiadis V. On Parallel Snapshot Isolation and Release/Acquire Consistency. In: Ahmed A, editor. Proceedings of the 27th European Sym-

posium on Programming. Cham: Lecture Notes in Computer Science; 2018. p. 940–967. pages 1

[9] Sovran Y, Power R, Aguilera MK, Li J. Transactional Storage for Geo-replicated Systems. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles. SOSP'11. New York, NY, USA: ACM; 2011. p. 385–400. Available from: `http://doi.acm.org/10.1145/2043556.2043592`. pages 1, 7

[10] Xiong S, Cerone A, Raad A, Gardner P. Data Consistency in Transactional Storage Systems: a Centralised Approach. arXiv preprint arXiv:190110615. 2019;. pages 1, 2, 3, 4, 8, 10, 11, 12, 13, 14, 15, 16, 19, 20, 23, 42, 45, 55, 60

[11] Adya A. Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions. Massachusetts Institute of Technology. Cambridge, MA, USA; 1999. Available from: `http://pmg.csail.mit.edu/papers/adya-phd.pdf`. pages 1, 6, 8

[12] Burckhardt S, Fahndrich M, Leijen D, Sagiv M. Eventually Consistent Transactions. In: Proceedings of the 21nd European Symposium on Programming. Springer; 2012. . pages 1, 8

[13] Cerone A, Bernardi G, Gotsman A. A Framework for Transactional Consistency Models with Atomic Visibility. In: Aceto L, de Frutos Escrig D, editors. 26th International Conference on Concurrency Theory (CONCUR 2015). vol. 42 of Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik; 2015. p. 58–71. Available from: `http://drops.dagstuhl.de/opus/volltexte/2015/5375`. pages 1, 8

[14] Crooks N, Pu Y, Alvisi L, Clement A. Seeing is Believing: A Client-Centric Specification of Database Isolation. In: Proceedings of the 2017 ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing. PODC'17. New York, NY, USA: ACM; 2017. p. 73–82. pages 1, 10

[15] Nagar K, Jagannathan S. Automated Detection of Serializability Violations Under Weak Consistency. In: Proceedings of the 29th International Conference on Concurrency Theory; 2018. p. 41:1–41:18. Available from: `https://doi.org/10.4230/LIPIcs.CONCUR.2018.41`. pages 1, 10

[16] Kaki G, Nagar K, Najafzadeh M, Jagannathan S. Alone Together: Compositional Reasoning and Inference for Weak Isolation. Proceedings of the ACM on Programming Languages. 2017 December;2(POPL):27:1–27:34. pages 1, 10

[17] Beillahi SM, Bouajjani A, Enea C. Checking Robustness Against Snapshot Isolation. CoRR. 2019;abs/1905.08406. Available from: `http://arxiv.org/abs/1905.08406`. pages 5

[18] Wickerson J, Batty M, Sorensen T, Constantinides GA. Automatically comparing memory consistency models. In: ACM SIGPLAN Notices. vol. 52. ACM; 2017. p. 190–204. pages 5, 9, 10

[19] Papadimitriou CH. The Serializability of Concurrent Database Updates. J ACM. 1979 October;26(4):631–653. pages 6

[20] Eswaran KP, Gray JN, Lorie RA, Traiger IL. The notions of consistency and predicate locks in a database system. Communications of the ACM. 1976;19(11):624–633. pages 6

[21] Gray JN, Lorie RA, Putzolu GR, Traiger IL. Readings in database systems. chapter Granularity of Locks and Degrees of Consistency in a Shared Data Base. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA; 1988. pages 6

[22] Alglave J, Maranget L, Tautschnig M. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. ACM Trans Program Lang Syst. 2014 July;36(2). Available from: `https://doi.org/10.1145/2627752`. pages 9

[23] Quick reference for litmus test families;. Available from: `https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test6.pdf`. pages 9

[24] Experimental validation of our axiomatic power model;. Available from: `http://moscova.inria.fr/~maranget/AXIOM/description.html`. pages 9

[25] Mador-Haim S, Alur R, Martin MM. Generating litmus tests for contrasting memory consistency models. In: International Conference on Computer Aided Verification. Springer; 2010. p. 273–287. pages 10

Appendix A

Command and Program Semantics

$$\textsc{TPrimitive}\ \dfrac{(s,\sigma)\overset{\mathtt{T}_p}{\rightsquigarrow}(s',\sigma') \qquad o = \mathsf{op}(s,\sigma,\mathtt{T}_p)}{(s,\sigma,\mathcal{F}),\mathtt{T}_p \rightsquigarrow (s',\sigma',\mathcal{F}\lll o),\mathtt{skip}}
\qquad
\begin{array}{lll}
\mathcal{F}\lll(\mathtt{R},k,v) & \overset{def}{=} & \begin{cases}\mathcal{F}\cup\{(\mathtt{R},k,v)\} & \text{if } \forall l,v'.\,(l,k,v')\notin\mathcal{F}\\ \mathcal{F} & \text{otherwise}\end{cases}\\
\mathcal{F}\lll(\mathtt{W},k,v) & \overset{def}{=} & (\mathcal{F}\setminus\{(\mathtt{W},k,v') \mid v'\in\textsc{Value}\})\cup\{(\mathtt{W},k,v)\}\\
\mathcal{F}\lll\epsilon & \overset{def}{=} & \mathcal{F}
\end{array}$$

$$\begin{array}{ll}
(s,\sigma)\overset{\mathtt{x:=E}}{\rightsquigarrow}(s[\mathtt{x}\mapsto[\![\mathtt{E}]\!]_s],\sigma) & (s,\sigma)\overset{\mathtt{assume(E)}}{\rightsquigarrow}(s,\sigma) \text{ where } [\![\mathtt{E}]\!]_s\neq 0\\
(s,\sigma)\overset{\mathtt{x:=[E]}}{\rightsquigarrow}(s[\mathtt{x}\mapsto\sigma([\![\mathtt{E}]\!]_s)],\sigma) & (s,\sigma)\overset{\mathtt{[E_1]:=E_2}}{\rightsquigarrow}(s,\sigma[[\![\mathtt{E}_1]\!]_s\mapsto[\![\mathtt{E}_2]\!]_s])
\end{array}$$

$$\begin{array}{ll}
\mathsf{op}(s,\sigma,\mathtt{x}:=\mathtt{E})\overset{def}{=}\epsilon & \mathsf{op}(s,\sigma,\mathtt{assume}(\mathtt{E}))\overset{def}{=}\epsilon\\
\mathsf{op}(s,\sigma,\mathtt{x}:=[\mathtt{E}])\overset{def}{=}(\mathtt{R},[\![\mathtt{E}]\!]_s,\sigma([\![\mathtt{E}]\!]_s)) & \mathsf{op}(s,\sigma,[\mathtt{E}_1]:=\mathtt{E}_2)\overset{def}{=}(\mathtt{W},[\![\mathtt{E}_1]\!]_s,[\![\mathtt{E}_2]\!]_s)
\end{array}$$

Figure A.1: The semantics of transactional commands [10]

$$\textsc{CPrimitive}\ \dfrac{s\overset{\mathtt{C}_p}{\rightsquigarrow}s'}{cl\vdash(\mathcal{K},u,s),\mathtt{C}_p\xrightarrow{(cl,\iota)}_{\mathtt{ET}}(\mathcal{K},u,s'),\mathtt{skip}}
\qquad
\begin{array}{l}
s\overset{\mathtt{x:=E}}{\rightsquigarrow}s[\mathtt{x}\mapsto[\![\mathtt{E}]\!]_s]\\
s\overset{\mathtt{assume(E)}}{\rightsquigarrow}s \text{ where } [\![\mathtt{E}]\!]_s\neq 0
\end{array}$$

$$\textsc{CAtomicTrans}\ \dfrac{\begin{array}{c}u\sqsubseteq u'' \quad \sigma=\mathsf{snapshot}(\mathcal{K},u'') \quad (s,\sigma,\emptyset),\mathtt{T}\rightsquigarrow^*(s',\_,\mathcal{F}),\mathtt{skip} \quad \mathtt{canCommit}_{\mathtt{ET}}(\mathcal{K},u'',\mathcal{F})\\ t\in\mathsf{NextTxID}(cl,\mathcal{K}) \qquad \mathcal{K}'=\mathsf{UpdateKV}(\mathcal{K},u'',\mathcal{F},t) \qquad \mathtt{vShift}_{\mathtt{ET}}(\mathcal{K},u'',\mathcal{K}',u')\end{array}}{cl\vdash(\mathcal{K},u,s),[\mathtt{T}]\xrightarrow{(cl,u'',\mathcal{F})}_{\mathtt{ET}}(\mathcal{K}',u',s'),\mathtt{skip}}$$

$$\textsc{PProg}\ \dfrac{u=\mathcal{U}(cl) \qquad s=\mathcal{E}(cl) \qquad \mathtt{C}=\mathtt{P}(cl) \qquad cl\vdash(\mathcal{K},u,s),\mathtt{C}\xrightarrow{\lambda}_{\mathtt{ET}}(\mathcal{K}',u',s'),\mathtt{C}'}{\vdash(\mathcal{K},\mathcal{U},\mathcal{E}),\mathtt{P}\xrightarrow{\lambda}_{\mathtt{ET}}(\mathcal{K}',\mathcal{U}[cl\mapsto u'],\mathcal{E}[cl\mapsto s']),\mathtt{P}[cl\mapsto\mathtt{C}']}$$

Figure A.2: The semantics of sequential commands and programs [10]