Data Consistency in Transactional Storage Systems: A Centralised Semantics

Abstract. We introduce an interleaving operational semantics for describing the client-observable behaviour of atomic transactions on distributed key-value stores. Our semantics builds on abstract states comprising centralised, global key-value stores and partial client views. We provide operational definitions of consistency models for our abstract states which we show to be equivalent to the well-known declarative definitions of consistency model for execution graphs. We explore two applications, verifying that the COPS geo-replicated database and the Clock-SI partitioned database satisfy their consistency models using trace refinement, and proving invariant properties of client programs.

1 Introduction

Transactions are the *de facto* synchronisation mechanism in modern distributed databases. To achieve scalability and performance, distributed databases often use weak transactional consistency guarantees. These weak guarantees pose two main challenges: the verification that database protocols satisfy a particular consistency guarantee; and the analysis of client behaviour with such guarantees.

These consistency guarantees are known as *consistency models* Much work has been done to formalise the semantics of such consistency models: declaratively, several *general* formalisms have been proposed, such as dependency graphs [1] and abstract execution [13], to provide a unified axiomatic semantics for formulating different consistency models; operationally, although specific consistency models have been captured using reference implementations [9, 45, 40, 35, 5, 39], there has been little work on *general* operational semantics for describing a range of consistency models. Our goal is to provide such a general operational semantics, to verify database protocols of particular consistency models and to analyse client programs with respect to such consistency models.

The declarative approach has been substantially studied [1, 13, 15, 18]. It uses a two-step procedure: (1) construct *candidate executions* of the whole program comprising transactions in which reads may contain an arbitrary value; and (2) apply a number of axioms on such executions to rule out invalid executions. Such axioms may state, for example, that every read is validated by a write that has written the read value. This axiomatic approach is unsuitable for invariant-based program analysis which requires reasoning about the step-wise execution of a program.

The operational approach has been less studied [37, 28, 19, 30]. Nagar and Jagannathan [37] introduce an operational semantics over abstract execution graphs, and prove robustness of client programs using model checking. Each

execution step adds a new transaction node in the graph by first constructing candidate steps and then applying axioms to the whole graph to rule out invalid candidate steps. This approach is unlikely to be suitable for invariant-based analysis associated with the state. Kaki et al. [28] propose an operational semantics for SQL transaction programs over an abstract centralised store with the consistency models given by the standard ANSI/SQL isolation levels [9]. They develop a program logic and prototype tool for reasoning about client programs, and so can capture invariant properties of the state. They can express snapshot isolation [9], but the consensus is that they cannot capture the weaker consistency models such as parallel snapshot isolation [45] important for distributed databases. Crooks et al. [19] provide a trace semantics over a global centralised store, where an execution step involves analysing the totally-ordered history of states and the read-write set of the transaction. They show the equivalence of several implementation-specific definitions of consistency model. However, their approach does not lend itself to analysing client programs, since observations made by clients require the total order in which transactions commit which is not realistic for clients. Koskinen and Parkinson [30] provide a log-based abstract operational semantics for verifying several implementations of serilisability. However, it is unknown that they could be easily extended to tackle weaker consistency models.

We introduce an *interleaving* operational semantics for describing the client-observable behaviour of atomic transactions on distributed key-value stores (§ 3). Our semantics uses abstract states comprising *a global, centralised key-value store* (kv-store) with *multi-versioning*, which records all the versions of a key, and *partial client views*, which let clients see only a subset of the versions. Our client views partly inspired by the views in the C11 operational semantics in [29]. An execution step depends simply on the abstract state, the read-write set of the transaction, and an *execution test* which determines if a client with a given view is allowed to commit a transaction. Different execution tests give rise to different consistency models, which we show to be equivalent to well-known declarative definitions of consistency model for execution graphs (§ 6). Our execution tests resemble the approach taken in [19], except that they require an analysis of the whole trace for an execution step whereas we require the current abstract state.

We make the assumption that our transactions satisfy the *snapshot property*, also known as *atomic visibility*. This means that the client observes that a transaction reads from an atomic snapshot of the database and commits atomically, even if the underlying distributed protocol has a more fine-grained behaviour. This assumption is reasonable in distributed databases: for example, with online shopping application, a client only sees one snapshot of the database and only has knowledge that their transaction has successfully committed. Our execution tests uniformly capture the well-known consistency models that satisfy this snapshot property (§ 4): e.g., causal consistency (`CC`) [35, 39], parallel snapshot isolation (`PSI`) [45, 40], snapshot isolation (`SI`) [9] and serialisability [9]. We also identify a new consistency model, called *weak snapshot isolation* (`WSI`), that interestingly sits between `PSI` and `SI` and retains good properties of both.

Using our operational semantics, we demonstrate that we can verify that database protocols satisfy particular consistency models and prove invariant properties of client programs with respect to such consistency models (§ 5). In particular, we establish the correctness of two database protocols: the COPS protocol for the fully replicated kv-stores [35] which satisfies causal consistency (§ 5.1); and the Clock-SI protocol for partitioned kv-stores [24] which satisfies snapshot isolation (§H). We prove invariant properties of client programs including general robustness results for client programs with certain patterns of transaction and invariant properties of specific programs not satisfying such robustness results (§ 5.2). We believe out robustness results are the first to take into account client sessions: with sessions, we show that multiple counters *are not* robust against PSI; interestingly, without sessions, Bernardi and Gotsman [10] show that multiple counters *are* robust against PSI using static-analysis techniques which are known not to be applicable to sessions.

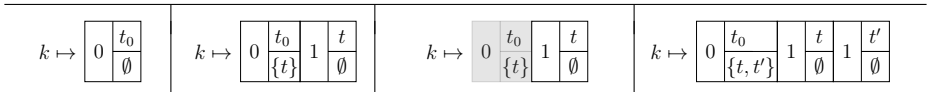
With our operational semantics, we believe that we have identified an interesting *mid-point* between distributed databases and clients. As far as we are aware, there has been no previous work on transactions that, in the same general semantics, verifies distributed protocols and analyses client programs. This was an important goal for us, resonating with recent work that does just this for standard shared-memory concurrency [22, 20, 27, 38].

2 Overview

We motivate our key ideas, centralised kv-stores, partial client views and execution tests, via an intuitive example. We show that our interleaving semantics is an ideal mid-point for verifying distributed protocols and proving invariant properties of client programs.

Example We use a simple transactional library, $\texttt{Counter}(k)$, to introduce our operational semantics. Clients of this counter library can manipulate the value of key k via two transactions: $\mathsf{inc}(k) \triangleq [\texttt{x} := [k]\,;\ [k] := \texttt{x}{+}1]$ and $\mathsf{read}(k) \triangleq [\texttt{x} := [k]]$. Command $\texttt{x} := [k]$ reads the value of key k to local variable x, and command $[k] := \texttt{x}{+}1$ writes the value of $\texttt{x}{+}1$ to key k. The code of each operation is wrapped in square brackets, denoting that it must be executed *atomically*.

Consider a replicated database where a client only interacts with one replica. For such a database, the correctness of atomic transactions is subtle, depending heavily on the particular consistency model under consideration. Consider the client program $\mathtt{P}_{\mathtt{LU}} \triangleq (cl_1 : \mathsf{inc}(k) \;||\; cl_2 : \mathsf{inc}(k))$, where we assume that the clients cl_1 and cl_2 work on different replicas and the k initially holds value 0 in all replicas. Intuitively, since transactions are executed atomically, after both calls to $\mathsf{inc}(k)$ have terminated, the counter should hold the value 2. Indeed, this is the only outcome allowed under serialisability (SER), where transactions appear to execute in a sequential (serial) order, one after another. The implementation of SER in distributed kv-stores comes at a significant performance cost. Therefore, implementers are content with weaker consistency models [7, 6, 5, 34, 35, 46, 32, 45, 42, 4, 24, 11]. For example, if the replicas provide no synchronisation mechanism for transactions, then it is possible for both clients to read the same



(a) Initial state (b) After t (c) A possible view of cl_2 (d) After t', lost update

Fig. 1: Example key-value stores (a, b, d); a client view (c)

initial value 0 for k at their distinct replicas, update them to 1, and eventually propagate their updates to other replicas. Thus, both sites are unchanged with value 1 for k. This weak behaviour is known as the *lost update* anomaly, which is allowed under the consistency model called *causal consistency* [35, 46, 32].

Centralised Operational Semantics A well-known declarative approach for a range of consistency models is to use execution graphs [2, 1, 15, 13], where nodes are atomic transactions and edges describe the known dependencies between transactions. The graphs capture the behaviour of the whole program, with different consistency models corresponding to different sets of axioms ruling out the invalid graphs. However, execution graphs provide little information about how the state of a kv-store evolves throughout the execution of a program. By contrast, we provide an interleaving operational semantics based on an abstract centralised state. The centralised state comprises a centralised, multi-versioned kv-store, which is *global* in the sense that it contains all the versions written by clients, and client views of the store, which are *partial* in the sense that clients may see different subsets of the versions in the kv-store. Each update is given by either a primitive command or an atomic transaction. The atomic transaction steps are subject to an *execution test* which analyses the state to determine if the update is allowed by the associated consistency model.

Let us introduce our global kv-stores and partial client views by showing that we can reproduce the lost update anomaly given by $\mathtt{P_{LU}}$. Our kv-stores are functions mapping keys to lists of versions, where the versions record all the values written to each key together with the meta-data of the transactions that access it. In the $\mathtt{P_{LU}}$ example, the initial kv-store comprises a single key k, with only one possible version $(0, t_0, \emptyset)$, stating that k holds value 0, that the version *writer* is the initialising transaction t_0 (this version was written by t_0), and that the version *reader set* is empty (no transaction has read this version). Fig. 1a depicts this initial kv-store, with the version represented as a box sub-divided in three sections: the value 0; the writer t_0; and the reader set $\emptyset$.

First, suppose that cl_1 invokes inc on Fig. 1a. It does this by choosing a fresh transaction identifier, t, and then proceeds with $\mathsf{inc}(k)$. It reads the initial version of k with value 0 and then writes a new value 1 for k. The resulting kv-store is depicted in Fig. 1b, where the initial version of k has been updated to reflect that it has been read by t.

Second, client cl_2 invokes inc on Fig. 1b. As there are now two versions available for k, we must determine the version from which cl_2 fetches its value, before running $\mathsf{inc}(k)$. This is where *client views* come into play. Intuitively, a view of client cl_2 comprises those versions in the kv-store that are *visible* to cl_2, i.e. those that can be read by cl_2. If more than one version is visible, then the newest (right-most) version is selected, modelling the *last-write-wins* resolution

policy used by many distributed kv-stores. In our example, there are two view candidates for cl_2 when running $\mathsf{inc}(k)$ on Fig. 1b: (1) one containing only the initial version of k; (2) the other containing both versions of k.[1] For (1), the view is depicted in Fig. 1c. Client cl_2 chooses a fresh transaction identifier t', reads the initial value 0 and writes a new version with value 1, as depicted in Fig. 1d. Such a kv-store does not contain a version with value 2, despite two increments on k, producing the lost update anomaly. For (2), client cl_2 reads the newest value 1 and writes a new version with value 2.

To avoid undesirable behaviour, such as the lost update anomaly, we use an *execution test* which restricts the possible update at the point of the transaction commit. One such test is to enforce a client to commit a transaction writing to k if and only if its view contains all versions available in the global state for k. This prevents cl_2 from running $\mathsf{inc}(k)$ on Fig. 1b if its view only contains the initial version of k. Instead, the cl_2 view must contain both versions of k, thus enforcing cl_2 to write a version with value 2 after running $\mathsf{inc}(k)$. This particular test corresponds to *write-conflict freedom*: at most one concurrent transaction can write to a key at any one time. In § 4, we give many examples of execution tests and their associated consistency models on kv-stores. In § 6 and §F, we show the equivalence of our operational definitions of consistency models and the declarative ones based on execution graphs.

Verifying Implementation Protocols The first application of our operational semantics is for showing that implementations of distributed key-value stores satisfy certain consistency models. Our abstract states provide a faithful abstraction of geo-replicated and partitioned databases, and execution tests provide a powerful abstraction of the synchronisation mechanisms enforced by these databases when committing a transaction. This then allows us to use our formalism to verify the correctness of distributed database protocols. To demonstrate this, we show that the COPS protocol [35] for implementing a replicated database satisfies causal consistency (§ 5.1 and §H.1), and the Clock-SI protocol [24] for implementing a partitioned database satisfies snapshot isolation (§H.2).

Proving Invariant Properties of Client Programs The second application of our operational semantics is to prove invariant properties of client programs (§ 5.2). One property is the robustness for client programs. A program `P` is robust if any kv-stores obtained by executing `P` under a weak consistency model can also be obtained under serialisability. To demonstrate this, we prove the robustness of the single counter library discussed above against `PSI`, and the robustness of a multi-counter library and the banking library of Alomari et al. [3] against our new proposed model `WSI` and hence all stronger models such as `SI`. The latter is done through general conditions on invariant which guarantees robustness against `WSI`. Apart from robustness, we show that a lock paradigm is correct under `UA`, although it is not robust. Thanks to our operational semantics, our invariant-based approaches only need to work with single program steps rather than whole program traces.

[1] As we explain in § 3.1, we always require the view of a client to include the initial version of each key.

3 Operational Model

We define an interleaving operational semantics for atomic transactions over global, centralised kv-stores and partial client views.

3.1 Key-Value Stores and Client Views

Our global, centralised key-value stores (kv-store) and partial client views provide the abstract machine states for our operational semantics. 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.

We assume a countably infinite set of *client identifiers*[2], $\textsc{ClientID} \ni cl$. The set of *transaction identifiers*, $\textsc{TransID} \ni t$, is defined by $\textsc{TransID} \triangleq \{t_0\} \uplus \{t^n_{cl} \mid cl \in \textsc{ClientID} \wedge n \geq 0\}$, where t_0 denotes the *initialisation transaction* and t^n_{cl} identifies a transaction committed by client cl with n determining the client session order: that is, $\mathsf{SO} \triangleq \{(t, t') \mid \exists cl, n, m.\ t = t^n_{cl} \wedge t' = t^m_{cl} \wedge n < m\}$. Subsets of $\textsc{TransID}$ are ranged over by $T, T', \cdots$. We let $\textsc{TransID}_0 \triangleq \textsc{TransID} \setminus \{t_0\}$.

Definition 1 (Kv-stores). *Assume a countably infinite set of* keys, $\textsc{Keys} \ni k$, *and a countably infinite set of* values, $\textsc{Val} \ni v$, *which includes the keys and an* initialisation value v_0. *The set of* versions, $\textsc{Version} \ni \nu$, *is defined by* $\textsc{Version} \triangleq \textsc{Val} \times \textsc{TransID} \times \mathcal{P}(\textsc{TransID}_0)$. *A* kv-store *is a function* $\mathcal{K} : \textsc{Keys} \to \mathsf{List}(\textsc{Version})$, *where* $\mathsf{List}(\textsc{Version}) \ni \mathcal{V}$ *is the set of lists of versions.*

Each version has 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 the notation $\mathsf{val}(\nu)$, $\mathsf{w}(\nu)$ 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 $\mathcal{K}$, write $|\mathcal{K}(k)|$ for the length of the version list $\mathcal{K}(k)$, and write $\mathcal{K}(k, i)$ for the i^{th} version of k.

We assume that the version list for each key has an initialisation version carrying the initialisation value v_0, written by the initialisation transaction t_0 with an initial empty reader set. We focus on kv-stores whose consistency model satisfies the *snapshot property*, ensuring that a transaction reads and writes at most one version for each key.

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

This is a normal assumption for distributed databases, e.g. in [5, 34, 35, 46, 32, 45, 42, 4, 24, 11]. Finally, we assume that the kv-store agrees with the session order of clients: 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.

$$\forall k, cl, i, j, n, m.\ (i < j \wedge t^n_{cl} = \mathsf{w}(\mathcal{K}(k, i)) \wedge t^m_{cl} \in \{\mathsf{w}(\mathcal{K}(k, j))\} \cup \mathsf{rs}(\mathcal{K}(k, i))) \Rightarrow n < m$$

A kv-store is *well-formed* if it satisfies these assumptions. Henceforth, we assume kv-stores are well-formed, and let KVS denote the set of well-formed kv-stores.

[2] We use the notation $\mathrm{A} \ni a$ to denote that elements of A are ranged over by a and its variants such as $a', a_1, \cdots$.

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. By contrast, in both replicated and partitioned distributed databases, a client may have incomplete information about updates distributed between machines. We model this incomplete information by defining a *view* of the kv-store which provides a *partial* record of the updates observed by a client. We require that a client view be *atomic* in that it can see either all or none of the updates of a transaction.

Definition 2 (Views). *A* view *of a kv-store* $\mathcal{K} \in \text{KVS}$ *is a function* $u \in \textsc{Views}(\mathcal{K}) \triangleq \textsc{Keys} \to \mathcal{P}(\mathbb{N})$ *such that, for all* i, i', k, k'*:*

$$0 \in u(k) \wedge (i \in u(k) \Rightarrow 0 \leq i < |\mathcal{K}(k)|) \qquad \text{(well-formed)}$$

$$i \in u(k) \wedge \mathsf{w}(\mathcal{K}(k,i)) = \mathsf{w}(\mathcal{K}(k',i')) \Rightarrow i' \in u(k') \qquad \text{(atomic)}$$

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

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

Definition 3 (Configurations). *A* configuration*,* $\Gamma \in \textsc{Conf}$*, is a pair* $(\mathcal{K}, \mathcal{U})$ *with* $\mathcal{K} \in \text{KVS}$ *and* $\mathcal{U} : \textsc{ClientID} \xrightarrow{fin} \textsc{Views}(\mathcal{K})$*. 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* $\mathcal{K}_0(k) \triangleq (v_0, t_0, \emptyset)$ *for all* $k \in \textsc{Keys}$*.*

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 $i, j \in u(k)$ and $i < j$, then 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 $\mathcal{K}(k,i)$. It is therefore possible 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. However, our formalism can be adapted straightforwardly to capture other resolution policies.

Definition 4 (View Snapshots). *Given* $\mathcal{K} \in \text{KVS}$ *and* $u \in \textsc{Views}(\mathcal{K})$*, the* snapshot *of* u *in* $\mathcal{K}$ *is a function,* $\mathsf{snapshot}(\mathcal{K}, u) : \textsc{Keys} \to \textsc{Val}$*, defined by* $\mathsf{snapshot}(\mathcal{K}, u) \triangleq \lambda k.\, \mathsf{val}(\mathcal{K}(k, \max_<(u(k))))$*, where* $\max_<(u(k))$ *is the maximum element in* $u(k)$ *with respect to the natural order* $<$ *over* $\mathbb{N}$*.*

3.2 Operational Semantics

Programming Language We assume a language of expressions built from values v and program variables $\mathtt{x}$, defined by: $\mathtt{E} ::= v \mid \mathtt{x} \mid \mathtt{E} + \mathtt{E} \mid \cdots$. The *evaluation* $[\![\mathtt{E}]\!]_s$ of expression $\mathtt{E}$ is parametric in the client-local stack s:

$$[\![v]\!]_s \triangleq v \quad [\![\mathtt{x}]\!]_s \triangleq s(\mathtt{x}) \quad [\![\mathtt{E}_1 + \mathtt{E}_2]\!]_s \triangleq [\![\mathtt{E}_1]\!]_s + [\![\mathtt{E}_2]\!]_s \quad \ldots$$

A *program* $\mathtt{P}$ comprises a finite number of clients, where each client is associated with a unique identifier $cl \in \textsc{ClientID}$, and executes a sequential *command* $\mathtt{C}$, given by the following grammar:

$$\mathtt{C} ::= \mathtt{skip} \mid \mathtt{C}_p \mid [\mathtt{T}] \mid \mathtt{C}\,;\mathtt{C} \mid \mathtt{C}+\mathtt{C} \mid \mathtt{C}^* \qquad \mathtt{C}_p ::= \mathtt{x} := \mathtt{E} \mid \mathtt{assume}\,(\mathtt{E})$$
$$\mathtt{T} ::= \mathtt{skip} \mid \mathtt{T}_p \mid \mathtt{T}\,;\mathtt{T} \mid \mathtt{T}+\mathtt{T} \mid \mathtt{T}^* \qquad \mathtt{T}_p ::= \mathtt{C}_p \mid \mathtt{x} := [\mathtt{E}] \mid [\mathtt{E}] := \mathtt{E}$$

Sequential commands $\mathtt{C}$ comprise $\mathtt{skip}$, primitive commands $\mathtt{C}_p$, atomic transactions $[\mathtt{T}]$, and standard compound constructs: sequential composition (;), non-deterministic choice (+) and iteration ($*$). Primitive commands include variable assignment $\mathtt{x} := \mathtt{E}$ and assume statements $\mathtt{assume}\,(\mathtt{E})$ which can be used to encode conditionals. They are used for computations based on client-local variables and can hence be invoked without restriction. Transactional commands $\mathtt{T}$ comprise $\mathtt{skip}$, primitive transactional commands $\mathtt{T}_p$, and the standard compound constructs. Primitive transactional commands comprise primitive commands $\mathtt{C}_p$, lookup $\mathtt{x} := [\mathtt{E}]$ and mutation $[\mathtt{E}] := \mathtt{E}$ used for reading and writing a single key to kv-stores respectively, which can only be invoked as part of an atomic transaction.

A *program* is a finite partial function from client identifiers to sequential commands. For clarity, we often write $\mathtt{C}_1 \parallel \ldots \parallel \mathtt{C}_n$ as syntactic sugar for a program $\mathtt{P}$ with n clients associated with identifiers $cl_1 \ldots cl_n$, where each client cl_i executes $\mathtt{C}_i$. Each client cl_i is associated with its own client-local *stack*, $s_i \in \textsc{Stack} \triangleq \textsc{Var} \to \textsc{Val}$, mapping program variables (ranged over by $\mathtt{x}, \mathtt{y}, \cdots$) to values.

Transactional Semantics In our operational semantics, transactions are executed *atomically*. It is still possible for an implementation, such as COPS [35], to update the underlying distributed key-value stores while the transaction is in progress. It just means that, given the abstractions captured by our global kv-stores and partial views, such an update is modelled as an instantaneous atomic update. Intuitively, given a configuration $\Gamma = (\mathcal{K}, \mathcal{U})$, when a client cl executes a transaction $[\mathtt{T}]$, it performs the following steps: (1) it constructs an initial *snapshot* σ of $\mathcal{K}$ using its view $\mathcal{U}(cl)$ as defined in Def. 4; (2) it executes $\mathtt{T}$ in isolation over σ accumulating the effects (the reads and writes) of executing $\mathtt{T}$; and (3) it commits $\mathtt{T}$ by incorporating these effects into $\mathcal{K}$.

Definition 5 (Transactional snapshots). *A* transactional snapshot*, $\sigma \in \textsc{Snapshot} \triangleq \textsc{Keys} \to \textsc{Val}$, is a function from keys to values. When the meaning is clear, it is just called a* snapshot*.*

The rules for transactional commands (Fig. 2) will be defined using an arbitrary transactional snapshot. The rules for sequential commands and programs (Fig. 3) will be defined using a transactional snapshot given by a view snapshot. To capture the effects of executing a transaction $\mathtt{T}$ on a snapshot σ of kv-store $\mathcal{K}$, we identify a *fingerprint* of $\mathtt{T}$ on σ which captures the values $\mathtt{T}$ reads from σ, and the values $\mathtt{T}$ writes to σ and intends to commit to $\mathcal{K}$. Execution of a transaction in a given configuration may result in more than one fingerprint due to non-determinism (non-deterministic choice).

$$\textsc{TPrimitive}\quad \frac{(s,\sigma) \overset{\mathtt{T}_p}{\rightsquigarrow} (s',\sigma') \qquad o = \mathsf{op}(s,\sigma,\mathtt{T}_p)}{(s,\sigma,\mathcal{F}),\mathtt{T}_p \to (s',\sigma',\mathcal{F} \lll o),\mathtt{skip}}$$

$$\mathcal{F} \lll (\mathtt{r},k,v) \triangleq \begin{cases} \mathcal{F} \cup \{(\mathtt{r},k,v)\} & \text{if } \forall l,v'.\ (l,k,v') \notin \mathcal{F} \\ \mathcal{F} & \text{otherwise} \end{cases}$$

$$\mathcal{F} \lll (\mathtt{w},k,v) \triangleq (\mathcal{F} \backslash \{(\mathtt{w},k,v') \mid v' \in \textsc{Val}\}) \cup \{(\mathtt{w},k,v)\}$$

$$\mathcal{F} \lll \epsilon \triangleq \mathcal{F}$$

Fig. 2: Rules for transactional commands

Definition 6 (Fingerprints). *Let* Ops *denote the set of* read ($\mathtt{r}$) and write ($\mathtt{w}$) operations *defined by* $\textsc{Ops} \triangleq \{(l,k,v) \mid l \in \{\mathtt{r},\mathtt{w}\} \wedge k \in \textsc{Keys} \wedge v \in \textsc{Val}\}$. *A* fingerprint $\mathcal{F}$ *is a set of operations,* $\mathcal{F} \subseteq \textsc{Ops}$, *such that:* $\forall k \in \textsc{Keys}, l \in \{\mathtt{r},\mathtt{w}\}.\ (l,k,v_1),(l,k,v_2) \in \mathcal{F} \Rightarrow v_1 = v_2$.

A fingerprint contains at most one read operation and at most one write operation for a given key. This reflects our assumption regarding transactions that satisfy the snapshot property: 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.

The rules for primitive transactional commands, TPrimitive, is given in Fig. 2. The rules for the compound constructs are straightforward and given in §A. The TPrimitive rule updates the snapshot and the fingerprint of a transaction: the premise $(s,\sigma) \overset{\mathtt{T}_p}{\rightsquigarrow} (s',\sigma')$ describes how executing $\mathtt{T}_p$ affects the local state (the client stack and the snapshot) of a transaction; and the premise $o = \mathsf{op}(s,\sigma,\mathtt{T}_p)$ identifies the operation on the kv-store associated with $\mathtt{T}_p$.

Definition 7. *The relation* $\overset{\mathtt{T}_p}{\rightsquigarrow} \subseteq (\textsc{Stack}\times\textsc{Snapshot})\times(\textsc{Stack}\times\textsc{Snapshot})$ *is defined by*[3] *:*

$$(s,\sigma) \overset{\mathtt{x:=E}}{\rightsquigarrow} (s[\mathtt{x} \mapsto [\![\mathtt{E}]\!]_s],\sigma) \qquad (s,\sigma) \overset{\mathtt{assume(E)}}{\rightsquigarrow} (s,\sigma) \text{ where } [\![\mathtt{E}]\!]_s \neq 0$$

$$(s,\sigma) \overset{\mathtt{x:=[E]}}{\rightsquigarrow} (s[\mathtt{x} \mapsto \sigma([\![\mathtt{E}]\!]_s)],\sigma) \qquad (s,\sigma) \overset{\mathtt{[E_1]:=E_2}}{\rightsquigarrow} (s,\sigma[[\![\mathtt{E}_1]\!]_s \mapsto [\![\mathtt{E}_2]\!]_s])$$

The function op*, computing the fingerprint of primitive commands, is defined by:*

$$\mathsf{op}(s,\sigma,\mathtt{x} := \mathtt{E}) \triangleq \epsilon \qquad \mathsf{op}(s,\sigma,\mathtt{assume}\,(\mathtt{E})) \triangleq \epsilon$$

$$\mathsf{op}(s,\sigma,\mathtt{x} := [\mathtt{E}]) \triangleq (\mathtt{r},[\![\mathtt{E}]\!]_s,\sigma([\![\mathtt{E}]\!]_s)) \qquad \mathsf{op}(s,\sigma,[\mathtt{E}_1] := \mathtt{E}_2) \triangleq (\mathtt{w},[\![\mathtt{E}_1]\!]_s,[\![\mathtt{E}_2]\!]_s)$$

The empty operation ϵ *is used for those primitive commands that do not contribute to the fingerprint.*

The conclusion of the TPrimitive rule uses the *combination operator* $\lll : \mathcal{P}(\textsc{Ops}) \times (\textsc{Ops} \uplus \{\epsilon\}) \to \mathcal{P}(\textsc{Ops})$, defined in Fig. 2, to extend the fingerprint $\mathcal{F}$ accumulated with operation o associated with $\mathtt{T}_p$, as appropriate: it adds a read from k if $\mathcal{F}$ contains no entry for k; and it always updates the write for k to $\mathcal{F}$, removing previous writes to k.

Command and Program Semantics We give the operational semantics of commands and programs in Fig. 3. The command semantics describes transitions of the form $cl \vdash (\mathcal{K},u,s),\mathtt{C} \xrightarrow{\lambda}_{\mathtt{ET}} (\mathcal{K}',u',s'),\mathtt{C}'$, stating that given the kv-store $\mathcal{K}$, view u and stack s, a client cl may execute command $\mathtt{C}$ for one step, updating the kv-store to $\mathcal{K}'$, the stack to s', the view to u' and the command to its

[3] For any function f, the new function $\mathsf{f}[\mathsf{d} \mapsto \mathsf{r}]$ is defined by $\mathsf{f}[\mathsf{d} \mapsto \mathsf{r}](\mathsf{d}) = \mathsf{r}$, and $\mathsf{f}[\mathsf{d} \mapsto \mathsf{r}](\mathsf{d}') = \mathsf{f}(\mathsf{d}')$ if $\mathsf{d}' \neq \mathsf{d}$.

$$\textsc{CPrimitive}\quad \frac{s \overset{\mathtt{C}_p}{\rightsquigarrow} s'}{cl \vdash (\mathcal{K}, u, s), \mathtt{C}_p \xrightarrow{(cl,\iota)}_{\mathtt{ET}} (\mathcal{K}, u, s'), \mathtt{skip}} \qquad s \overset{\mathtt{x:=E}}{\rightsquigarrow} s\big[\mathtt{x} \mapsto [\![\mathtt{E}]\!]_s\big]$$

$$s \overset{\mathtt{assume(E)}}{\rightsquigarrow} s \text{ where } [\![\mathtt{E}]\!]_s \neq 0$$

$$\textsc{CAtomicTrans}\quad \frac{\begin{array}{c} u \sqsubseteq u'' \quad \sigma = \mathsf{snapshot}\big(\mathcal{K}, u''\big) \quad (s, \sigma, \emptyset), \mathtt{T} \rightarrow^* (s', \_, \mathcal{F}), \mathtt{skip} \quad \mathsf{can\text{-}commit}_{\mathtt{ET}}(\mathcal{K}, u'', \mathcal{F}) \\ t \in \mathsf{nextTid}(cl, \mathcal{K}) \qquad \mathcal{K}' = \mathsf{update}\big(\mathcal{K}, u'', \mathcal{F}, t\big) \qquad \mathsf{vshift}_{\mathtt{ET}}(\mathcal{K}, u'', \mathcal{K}', u') \end{array}}{cl \vdash (\mathcal{K}, u, s), \big[\mathtt{T}\big] \xrightarrow{(cl,u'',\mathcal{F})}_{\mathtt{ET}} (\mathcal{K}', u', s'), \mathtt{skip}}$$

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

Fig. 3: Rules for sequential commands and programs

continuation $\mathtt{C}'$. The label λ is either of the form (cl, ι) denoting that cl executed a primitive command that required no access to $\mathcal{K}$, or $(cl, u'', \mathcal{F})$ denoting that cl committed an atomic transaction with final fingerprint $\mathcal{F}$ under the view u''. The semantics is parametric in the choice of *execution test* ET for kv-stores, which is used to generate the *consistency model* on kv-stores under which a transaction can execute. In § 4, we give many examples of execution tests for well-known consistency models. In § 6 and §F, we prove that our execution tests generate consistency models that are equivalent to the existing definitions of consistency models (using execution graphs).

The rules for the compound constructs are straightforward and given in §A. The rule for primitive commands, CPrimitive, depends on the transition system $\overset{\mathtt{C}_p}{\rightsquigarrow} \subseteq \textsc{Stack} \times \textsc{Stack}$ which simply describes how the primitive command $\mathtt{C}_p$ affects the local state of a client. The rule CAtomicTrans describes the execution of an atomic transaction under the execution test ET.

We explain the CAtomicTrans rule in detail. The first premise states that the current view u of the executing command may be advanced to a newer view u'' (see Def. 2). Given the new view u'', the transaction obtains a snapshot σ of the kv-store $\mathcal{K}$, and executes T locally to completion (skip), updating the stack to s', while accumulating the fingerprint $\mathcal{F}$; this behaviour is modelled in the second and third premises of CAtomicTrans. Note that the resulting snapshot is ignored as the effect of the transaction is recorded in the fingerprint $\mathcal{F}$. The $\mathsf{can\text{-}commit}_{\mathtt{ET}}(\mathcal{K}, u'', \mathcal{F})$ premise ensures that under the execution test ET, the final fingerprint $\mathcal{F}$ of the transaction is compatible with the (original) kv-store $\mathcal{K}$ and the client view u'', and thus the transaction *can commit*. Note that the can-commit check is parametric in the execution test ET. This is because the conditions checked upon committing depend on the consistency model under which the transaction is to commit. In § 4, we define can-commit for several execution tests associated with well-known consistency models.

Now we are ready for client cl to commit the transaction resulting in the kv-store $\mathcal{K}'$ with the client view u'' *shifting* to a new view u': (1) pick a fresh transaction identifier $t \in \mathsf{nextTid}(cl, \mathcal{K})$; (2) compute the new kv-store $\mathcal{K}' = \mathtt{update}(\mathcal{K}, u'', \mathcal{F}, t)$; and (3) check if the *view shift* is permitted under execution test ET using $\mathsf{vshift}_{\mathtt{ET}}(\mathcal{K}, u'', \mathcal{K}', u')$. Observe that as with can-commit, the vshift check is parametric in the execution test ET. Once again this is because the conditions

checked for shifting the client view depend on the consistency model. In § 4, we define vshift for several execution tests associated with well-known consistency models. The set $\mathsf{nextTid}(cl, \mathcal{K})$ is defined by $\mathsf{nextTid}(cl, \mathcal{K}) \triangleq \{t^n_{cl} \mid \forall m.\ t^m_{cl} \in \mathcal{K} \Rightarrow m < n\}$. The function $\mathsf{update}(\mathcal{K}, u, \mathcal{F}, t)$ describes how the fingerprint $\mathcal{F}$ of transaction t executed under view u updates kv-store $\mathcal{K}$: 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; for each write $(\mathtt{w}, k, v)$, it appends a new version $(v, t, \emptyset)$ to $\mathcal{K}(k)$.

Definition 8 (Transactional update). *The function* $\mathsf{update}(\mathcal{K}, u, \mathcal{F}, t)$*, is defined by:* $\mathsf{update}(\mathcal{K}, u, \emptyset, t) \triangleq \mathcal{K}$ *and*

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

$$\mathsf{update}(\mathcal{K}, u, \{(\mathtt{w}, k, v)\} \uplus \mathcal{F}, t) \triangleq \textit{let } \mathcal{K}' = \mathcal{K}[k \mapsto \mathcal{K}(k)::(v, t, \emptyset)] \textit{ in } \mathsf{update}(\mathcal{K}', u, \mathcal{F}, t)$$

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

The last rule, PProg (Fig. 3), captures the execution of a program step given a *client environment* $\mathcal{E} \in$ CEnv. A client environment $\mathcal{E}$ is a function from client identifiers to stacks, associating each client with its stack. We assume that the domain of client environment contains the domain of the program throughout the execution: $\mathrm{dom}(\mathtt{P}) \subseteq \mathrm{dom}(\mathcal{E})$. Program transitions are simply defined in terms of the transitions of their constituent client commands. This yields an interleaving semantics for transactions of different clients: a client executes a transaction in an atomic step without interference from the other clients.

4 Consistency Models: Kv-stores

We define what it means for a kv-store to be in a consistent state. Many different consistency models for distributed databases have been proposed in the literature [12, 34, 32, 45, 9], capturing different trade-offs between performance and application correctness: examples range from *serialisability*, a strong consistency model which only allows kv-stores obtained from a serial execution of transactions with inevitable performance drawbacks, to *eventual consistency*, a weak consistency model which imposes few conditions on the structure of a kv-store leading to good performance but anomalous behaviour. We define consistency models for our kv-stores, by introducing the notion of *execution test* which specifies whether a client is allowed to commit a transaction in a given kv-store. Each execution test 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 the execution test are satisfied. We explore a range of execution tests associated with well-known consistency models in the literature. In § 6 and §F, we demonstrate that our operational formulation of consistency models over kv-stores using execution tests are equivalent to the established declarative definitions of consistency models over abstract executions [13, 15].

Definition 9 (Execution tests). *An* execution test *is a set of tuples* $\mathtt{ET} \subseteq$ KVS×Views×Fp×KVS×Views *such that, for all* $(\mathcal{K}, u, \mathcal{F}, \mathcal{K}', u') \in \mathtt{ET}$*: (1)* $u \in$

ET	$\mathsf{can\text{-}commit}_{\mathtt{ET}}(\mathcal{K},u,\mathcal{F}) \triangleq \mathtt{closed}(\mathcal{K},u,R_{\mathtt{ET}})$	$\mathsf{vshift}_{\mathtt{ET}}(\mathcal{K},u,\mathcal{K}',u')$
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	$R_{\mathtt{CC}} \triangleq \mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}$	$\mathsf{vshift}_{\mathtt{MR}\cap\mathtt{RYW}}(\mathcal{K},u,\mathcal{K}',u')$
UA	$R_{\mathtt{UA}} \triangleq \bigcup_{(\mathsf{w},k,\_)\in\mathcal{F}} \mathsf{WW}_{\mathcal{K}}^{-1}(k)$	true
PSI	$R_{\mathtt{PSI}} \triangleq R_{\mathtt{UA}} \cup R_{\mathtt{CC}} \cup \mathsf{WW}_{\mathcal{K}}$	$\mathsf{vshift}_{\mathtt{MR}\cap\mathtt{RYW}}(\mathcal{K},u,\mathcal{K}',u')$
CP	$R_{\mathtt{CP}} \triangleq \mathsf{SO}; \mathsf{RW}_{\mathcal{K}}^{?} \cup \mathsf{WR}_{\mathcal{K}}; \mathsf{RW}_{\mathcal{K}}^{?} \cup \mathsf{WW}_{\mathcal{K}}$	$\mathsf{vshift}_{\mathtt{MR}\cap\mathtt{RYW}}(\mathcal{K},u,\mathcal{K}',u')$
SI	$R_{\mathtt{SI}} \triangleq R_{\mathtt{UA}} \cup R_{\mathtt{CP}} \cup (\mathsf{WW}_{\mathcal{K}}; \mathsf{RW}_{\mathcal{K}})$	$\mathsf{vshift}_{\mathtt{MR}\cap\mathtt{RYW}}(\mathcal{K},u,\mathcal{K}',u')$
SER	$R_{\mathtt{SER}} \triangleq \mathsf{WW}^{-1}$	true

Fig. 4: Execution tests of consistency models, where SO is as given in § 3.1.

VIEWS $(\mathcal{K})$ *and* $u' \in$ VIEWS $(\mathcal{K}')$*; (2)* $\mathsf{can\text{-}commit}_{\mathtt{ET}}(\mathcal{K},u,\mathcal{F})$*; (3)* $\mathsf{vshift}_{\mathtt{ET}}(\mathcal{K},u,\mathcal{K}',u')$*; and (4) for all* $k \in \mathcal{K}$ *and* $v \in$ VAL*, if* $(\mathtt{r},k,v) \in \mathcal{F}$ *then* $\mathcal{K}(k, \max_{<}(u\,(k))) = v$.

Intuitively, $(\mathcal{K},u,\mathcal{F},\mathcal{K}',u') \in \mathtt{ET}$ means 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 $\mathcal{K}'$ (Def. 8) while shifting its view to u'. We adopt the notation $\mathtt{ET} \vdash (\mathcal{K},u) \triangleright \mathcal{F} : (\mathcal{K}',u')$ to capture this intuition. Note that the last condition in Def. 9 enforces the last-write-wins policy [48]: a transaction always reads the most recent writes from the initial view u.

Definition 10 (Consistency model). *The* consistency model *induced by an execution test* ET *is defined as* $\mathsf{CM}(\mathtt{ET}) \triangleq \{\mathcal{K} \mid \exists \Gamma_0 \in \text{CONF}_0, \mathtt{P}.\ \Gamma_0, \mathtt{P} \xrightarrow{\cdot}{}^{*}_{\mathtt{ET}} (\mathcal{K}, \_), \_\}$.

The largest execution test is denoted by $\mathtt{ET}_{\top}$, where for all $\mathcal{K},\mathcal{K}',u,u,\mathcal{F}$:

$$\mathsf{can\text{-}commit}_{\mathtt{ET}_{\top}}(\mathcal{K},u,\mathcal{F}) \overset{\text{def}}{\Leftrightarrow} \mathsf{true} \qquad \text{and} \qquad \mathsf{vshift}_{\mathtt{ET}_{\top}}(\mathcal{K},u,\mathcal{K}',u') \overset{\text{def}}{\Leftrightarrow} \mathsf{true}$$

The consistency model induced by $\mathtt{ET}_{\top}$ corresponds to the *Read Atomic* [5], a variant of *Eventual Consistency* [13] for atomic transactions. We give many examples of execution tests in the following sub-section.

4.1 Example Execution Tests

We give several examples of execution tests which give rise to consistency models on kv-stores. Recall that the snapshot property and the last write wins are hard-wired into our model. This means that we can only define consistency models that satisfy these two constraints. Although this forbids us to express interesting consistency models such as *Read Committed*, we are able to express a large variety of consistency models employed by distributed kv-stores.

Notation Given relations $\mathsf{r},\mathsf{r}' \subseteq \mathrm{A} \times \mathrm{A}$, 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 $\{(a_1,a_2) \mid \exists a.\ (a_1,a) \in \mathsf{r} \wedge (a,a_2) \in \mathsf{r}'\}$.

Recall that an execution test ET (Def. 9) has the form $(\mathcal{K},u,\mathcal{F},\mathcal{K}',u')$ where $\mathsf{can\text{-}commit}_{\mathtt{ET}}(\mathcal{K},u,\mathcal{F})$ and $\mathsf{vshift}_{\mathtt{ET}}(\mathcal{K},u,\mathcal{K}',u')$. We define can-commit and vshift for several consistency models, using some auxiliary definitions.

Prefix Closure Given a kv-store $\mathcal{K}$ and a view u, the *set of visible transactions* is $\mathsf{visTx}(\mathcal{K}, u) \triangleq \{\mathsf{w}(\mathcal{K}(k,i)) \mid i \in u\,(k)\}$. Given a binary relation on transactions, $R \subseteq \textsc{TransID} \times \textsc{TransID}$, we say that a view u is closed with respect to a kv-store $\mathcal{K}$ and R, written $\texttt{closed}(\mathcal{K}, u, R)$, iff:

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

That is, if transaction t is visible in u as $t \in \mathsf{visTx}(\mathcal{K}, u)$, then all the transactions t' that are R^*-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. $t' \in \mathsf{visTx}(\mathcal{K}, u)$.

Dependency Relations We define transaction dependency relations for 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 we say that, for every key k, there is:

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

Fig. 5a illustrates an example kv-store and its transaction dependency relations. We adopt the same names as the dependency relations for dependency graphs [1] to emphasise the similarity. However, the relations here do *not* depend on those in dependency graphs.

We give several definitions of execution tests using vshift and $\mathsf{can\text{-}commit}$ in Fig. 4. In § 6 and §F, we show that our definitions correspond to the well-known declarative definitions of consistency models on abstract executions.

Monotonic Reads (MR) This consistency model states that when committing, a client cannot lose information in that it can only see increasingly more up-to-date versions from a kv-store. This prevents, for example, the kv-store of Fig. 5b, since client cl first reads the latest version of k in t^1_{cl}, and then reads the older, initial version of k in t^2_{cl}. As such, the $\mathsf{vshift}_{\texttt{MR}}$ predicate in Fig. 4 ensures that clients can only extend their views. When this is the case, clients can *always* commit their transactions, and thus $\mathsf{can\text{-}commit}_{\texttt{MR}}$ is simply defined as `true`.

Read Your Writes (RYW) This consistency model states that a client must always see all the versions written by the client itself. The $\mathsf{vshift}_{\texttt{RYW}}$ 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. Note that under RYW the kv-store in Fig. 5c is prohibited as the initial version of k holds value v_0 and client cl tries to update the value of k twice. For its first transaction t^1_{cl}, it reads the initial value v_0 and then writes a new version with value v_1. For its second transaction t^2_{cl}, it reads again the initial value v_0 again and write a new version with value v_1. The $\mathsf{vshift}_{\texttt{RYW}}$ predicate rules out this example by requiring that the client view, after it commits the transaction t^1_{cl}, includes the version it wrote. When this is the case, clients can always commit their transactions, and thus $\mathsf{can\text{-}commit}_{\texttt{RYW}}$ is simply `true`.

The MR and RYW models together with *monotonic writes* (MW) and *write follows reads* (WFR) models are collectively known as *session guarantees*. Due to space constraints, the definitions associated with MW and WFR are in §A.

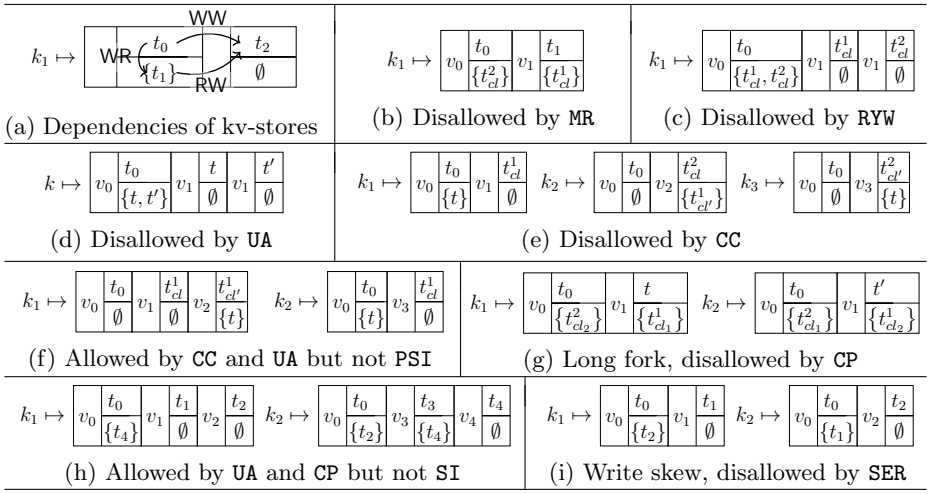


Fig. 5: Behaviours disallowed under different consistency models. Sub-figure 5a shows the dependencies of transactions in kv-stores (values omitted).

Causal Consistency (CC) Causal consistency subsumes the four session guarantees discussed above. As such, the $\mathsf{vshift}_{\mathtt{CC}}$ predicate is defined as the *conjunction* of their associated vshift predicates. However, as shown in Fig. 4, it is sufficient to define $\mathsf{vshift}_{\mathtt{CC}}$ as the conjunction of the MR and RYW session guarantees alone, where for brevity we write $\mathsf{vshift}_{\mathtt{MR}\cap\mathtt{RYW}}$ for $\mathsf{vshift}_{\mathtt{MR}} \wedge \mathsf{vshift}_{\mathtt{RYW}}$. This is because as we demonstrate in §A, the $\mathsf{vshift}_{\mathtt{MW}}$ and $\mathsf{vshift}_{\mathtt{WFR}}$ are defined simply as `true`, allowing us to remove them from $\mathsf{vshift}_{\mathtt{CC}}$.

Additionally, CC strengthens the session guarantees by requiring that if a client sees a version ν prior to committing a transaction, then it must also see the versions on which ν depends. If t is the writer of ν, then ν clearly depends on all versions that t reads. Moreover, if ν is, or it depends on, a version ν' accessed by a client cl, then it also depends on all versions that were previously read or written by cl. This is captured by the $\mathsf{can\text{-}commit}_{\mathtt{CC}}$ predicate in Fig. 4, defined as $\mathtt{closed}(\mathcal{K}, u, R_{\mathtt{CC}})$ with $R_{\mathtt{CC}} \triangleq \mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}$. For example, the kv-store of Fig. 5e is disallowed by CC: the version of key k_3 carrying value v_3 depends on the version of key k_1 carrying value v_1. However, transaction t must have been committed by a client whose view included v_3 of k_3, but not v_1 of k_1.

Update Atomic (UA) This consistency model has been proposed by Cerone et al. [15] and implemented by Liu et al. [34]. UA disallows concurrent transactions writing to the same key, a property known as *write-conflict freedom*: when two transactions write to the same key, one must see the version written by the other. Write-conflict freedom is enforced by $\mathsf{can\text{-}commit}_{\mathtt{UA}}$ which allows a client to write to key k only if its view includes all versions of k; i.e. its view is closed with respect to the $\mathsf{WW}^{-1}(k)$ relation for all keys k written in the fingerprint $\mathcal{F}$. This prevents the kv-store of Fig. 5d, as t and t' concurrently increment the initial version of k by 1. As client views must include the initial versions, once t commits a new version ν with value v_1 to k, then t' must include ν in its view

as there is a WW edge from the initial version to ν. As such, when t' increments k, it must read from ν, and not the initial version as depicted in Fig. 5d.

Parallel Snapshot Isolation (PSI) This consistency model is defined as the conjunction of the guarantees provided by CC and UA [15]. As such, the $\mathsf{vshift}_{\mathtt{PSI}}$ predicate is defined as the conjunction of the vshift predicates for CC and UA. However, we cannot simply define $\mathsf{can\text{-}commit}_{\mathtt{PSI}}$ as the conjunction of the can-commit predicates for CC and UA. This is for two reasons. First, their conjunction would only mandate that u be closed with respect to $R_{\mathtt{CC}}$ and $R_{\mathtt{UA}}$ *individually*, but *not* with respect to their *union*. Recall that closure is defined in terms of the transitive closure of a given relation and thus the closure of $R_{\mathtt{CC}}$ and $R_{\mathtt{UA}}$ is smaller than the closure of $R_{\mathtt{CC}} \cup R_{\mathtt{UA}}$. As such, we define $\mathsf{can\text{-}commit}_{\mathtt{PSI}}$ as closure with respect to $R_{\mathtt{PSI}}$ that must include $R_{\mathtt{CC}} \cup R_{\mathtt{UA}}$. Second, recall that $\mathsf{can\text{-}commit}_{\mathtt{UA}}$ requires that a transaction writing to a key k must be able to see all previous versions of k, i.e. all versions of k. That is, when write-conflict freedom is enforced, a version ν of k depends on all previous versions of k. This observation leads us to include write-write dependencies ($\mathsf{WW}_{\mathcal{K}}$) in $R_{\mathtt{PSI}}$. Observe that the kv-store in Fig. 5f shows an example kv-store that satisfies $\mathsf{can\text{-}commit}_{\mathtt{CC}} \wedge \mathsf{can\text{-}commit}_{\mathtt{UA}}$, but not $\mathsf{can\text{-}commit}_{\mathtt{PSI}}$.

Consistent Prefix (CP) If the total order in which transactions commit is known, CP can be described as a strengthening of CC: if a client sees the versions written by a transaction t, then it must also see all versions written by transactions that *commit* before t. Although kv-stores only provide *partial* information about the transaction commit order via the dependency relations, this is sufficient to formalise *Consistent Prefix* [18].

In practice, we can approximate the order in which transactions commit via the $\mathsf{WR}_{\mathcal{K}}, \mathsf{WW}_{\mathcal{K}}, \mathsf{RW}_{\mathcal{K}}$ and SO relations. This approximation is best understood in terms of an idealised implementation of CP on a centralised system, where the snapshot of a transaction is determined at its *start point* and its effects are made visible to future transactions at its *commit point*. With respect to this implementation, if $(t, t') \in \mathsf{WR}$, then t must commit before t' starts, and hence before t' commits. Similarly, if $(t, t') \in \mathsf{SO}$, then t commits before t' starts, and thus before t' commits. Recall that, if $(t'', t') \in \mathsf{RW}$, then t'' reads a version that is later overwritten by t'. That is, t'' cannot see the write of t', and thus t'' must start before t' commits. As such, if t commits before t'' starts ($(t, t'') \in \mathsf{WR}$ or $(t, t'') \in \mathsf{SO}$), and $(t'', t') \in \mathsf{RW}$, then t must commit before t' commits. In other words, if $(t, t') \in \mathsf{WR}; \mathsf{RW}$ or $(t, t') \in \mathsf{SO}; \mathsf{RW}$, then t commits before t'. Finally, if $(t, t') \in \mathsf{WW}$, then t must commit before t'. We therefore define $R_{\mathtt{CP}} \triangleq (\mathsf{WR}_{\mathcal{K}}; \mathsf{RW}^{?}_{\mathcal{K}} \cup \mathsf{SO}; \mathsf{RW}^{?}_{\mathcal{K}} \cup \mathsf{WW})$, approximating the order in which transactions commit. Cerone et al. [18] show that the set $(R^{-1}_{\mathtt{CP}})^{+}(t)$ contains all transactions that must be observed by t under CP. We define $\mathsf{can\text{-}commit}_{\mathtt{CP}}$ by requiring that the client view be closed with respect to $R_{\mathtt{CP}}$.

Consistent prefix disallows the *long fork anomaly* shown in Fig. 5g, where clients cl_1 and cl_2 observe the updates to k_1 and k_2 in different orders. Assuming without loss of generality that $t^2_{cl_1}$ commits before $t^2_{cl_2}$, then prior to committing its transaction cl_2 sees the version of k_1 with value v_0. However,

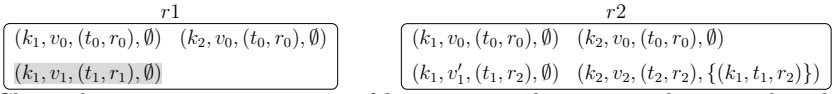
since $t \xrightarrow{\mathsf{WR}_{\mathcal{K}}} t^1_{cl_1} \xrightarrow{\mathsf{SO}} t^2_{cl_1} \xrightarrow{\mathsf{RW}} t' \xrightarrow{\mathsf{WR}} t^1_{cl_2}$, and prior to commit, $t^2_{cl_2}$ must see versions written by $t^1_{cl_2}$, then $t^2_{cl_2}$ should also see the version of k_1 with value v_2, leading to a contradiction.

Snapshot Isolation (SI) When the total order in which transactions commit is known, SI can be defined compositionally from CP and UA. As such, $\mathsf{vshift}_{\mathtt{SI}}$ is defined as the conjunction of their associated vshift predicates. However, as with PSI, we cannot define $\mathsf{can\text{-}commit}_{\mathtt{SI}}$ as the conjunction of their associated can-commit predicates. Rather, we define $\mathsf{can\text{-}commit}_{\mathtt{SI}}$ as closure with respect to $R_{\mathtt{SI}}$, which includes $R_{\mathtt{CP}} \cup R_{\mathtt{UA}}$. Observe that the kv-store in Fig. 5h shows an example kv-store that satisfies $\mathsf{can\text{-}commit}_{\mathtt{UA}} \wedge \mathsf{can\text{-}commit}_{\mathtt{CP}}$, but not $\mathsf{can\text{-}commit}_{\mathtt{SI}}$. Additionally, we include $\mathsf{WW};\mathsf{RW}$ in $R_{\mathtt{SI}}$. This is because when the centralised CP implementation (discussed before) is strengthened with write-conflict freedom, then a write-write dependency between two transactions t and t' does not only mandate that t commits before t' commits but also before t' starts. Consequently, if $(t, t') \in \mathsf{WW};\mathsf{RW}$, then t must commit before t' commit.

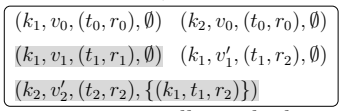
(Strict) serialisability (SER) Serialisability is the strongest consistency model in any framework that abstracts from aborted transactions, requiring that transactions execute in a total sequential order. The $\mathsf{can\text{-}commit}_{\mathtt{SER}}$ thus allows clients to commit transactions only when their view of the kv-store is complete, i.e. the client view is closed with respect to WW^{-1}. This requirement prevents the kv-store in Fig. 5i: without loss of generality, suppose that t_1 commits before t_2. Then the client committing t_2 must see the version of k_1 written by t_1, and thus cannot read the outdated value v_0 for k_1.

Weak Snapshot Isolation (WSI): A New Consistency Model Kv-stores and execution tests are useful for investigating new consistency models. One example is the consistency model induced by combining CP and UA, which we refer to as *Weak Snapshot Isolation* (WSI). To justify this consistency model in full, it would be useful to explore its implementations. Here we focus on the benefits of clients on WSI. Because WSI is stronger than CP and UA by definition, it forbids all the anomalies forbidden by these consistency models, e.g. the long fork (Fig. 5g) and the lost update (Fig. 5d). Moreover, WSI is strictly weaker than SI. As such, WSI allows all SI anomalies, e.g. the write skew (Fig. 5i), and allows behaviour not allowed under SI such as that in Fig. 5h. The kv-store $\mathcal{K}$ is reachable by executing transactions t_1, t_2, t_3 and t_4 in this order. In particular, t_4 is executed using $u{=}\{k_1 \mapsto \{0\}, k_2 \mapsto \{0, 1\}\}$. However, the same kv-store is not reachable under $\mathtt{ET}_{\mathtt{SI}}$. Under SI transaction t_4 cannot be executed using u: t_4 reads the version of k_2 written by t_3, but since $(t_2, t_3) \in \mathsf{RW}$ and $(t_1, t_2) \in \mathsf{WW}$, then u should contain the version of k_1 written by t_1, contradicting the fact that t_4 reads the initial version of k_1.

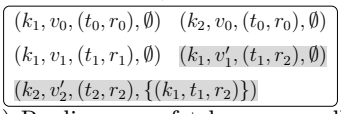
As WSI is a model weaker than SI, we believe that WSI implementations would outperform known SI implementations. Nevertheless, the two consistency models are very similar in that many applications that are correct under SI are also correct under WSI. We give examples of such applications in § 5.



(a) Client cl_1 commits a new version of k_1 carrying value v_1 to replica r_1; other clients commit versions to r_2. The new versions in r_1 and r_2 have not yet been synchronised.



(b) Replica r_1 optimistically reads the newest versions for k_1, k_2, one by one, during which time it receives synchronisation messages from r_2.



(c) Replica r_1 re-fetches a causally consistent snapshot for k_1, k_2 using the dependency sets.

Fig. 6: COPS protocol

5 Applications

To show the applications of our operational semantics, we use it to verify several distributed protocols (§ 5.1) and prove the invariants of transactional libraries (§ 5.2).

5.1 Application: Verifying Database Protocols

Kv-stores and views faithfully abstract the state of geo-replicated and partitioned databases, and execution tests provide a powerful abstraction of the synchronisation mechanisms enforced by these databases when committing a transaction. This makes it possible to use our semantics to verify the correctness of distributed database protocols. We demonstrate this by showing that the replicated database, COPS [35], satisfies causal consistency and the partitioned database, Clock-SI [24], satisfies snapshot isolation. We present an intuitive account of how we verify the COPS protocol using trace refinement. We refer the reader to §H.1 for the full details. In §H.2, we apply the same method to verify Clock-SI.

COPS Protocol COPS is a fully replicated database, with each replica storing multiple versions of each key as shown in Fig. 6a. Each COPS version ν such as $(k_1, v_1, (t_1, r_1), \emptyset)$ in Fig. 6a, contains a key (k_1), a value (v_1), a *unique* time-stamp (t_1, r_1) denoting when a client first wrote the version to the replica, and a set of dependencies $(\emptyset)$, written $\mathsf{deps}(\nu)$. The time-stamp associated with a version ν has the form (t, r), where r identifies the replica that committed ν, and t denotes the local time when r committed ν. Each dependency in $\mathsf{deps}(\nu)$ comprises a key and the time-stamp of the versions on which ν directly depends. We define the DEP relation, $(t, r) \xrightarrow{\mathsf{DEP}} (t', r')$, to denote that version (t, r) is included in the dependency set of version (t', r'). COPS assumes a total order over replica identifiers. As such, versions can be totally ordered lexicographically.

The COPS API provides two operations: one for writing to a *single* key; and another for atomically reading from *multiple* keys. Each call to a COPS operation is processed by a single replica. Each client maintains a *context*, which is a set of dependencies tracking the versions the client observes. We demonstrate how a client cl interacts with a replica through the program $\mathtt{P}_{\mathtt{COPS}}$:

$$\mathtt{P}_{\mathtt{COPS}} \triangleq (cl : [[k_1] := v_1]\,;\ [\mathtt{x} := [k_1]\,;\ \mathtt{y} := [k_2]])$$

For brevity, we assume that there are two keys, k_1 and k_2, and two replicas, r_1 and r_2, where $r_1 < r_2$ (Fig. 6a). Initially, client cl connects to replica r_1 and initialises its local context as $ctx{=}\emptyset$. To execute its first single-write transaction, cl requests to write v_1 to k_1 by sending the message (k_1, v_1, ctx) to its associated replica r_1 and awaits a reply. Upon receiving the message, r_1 produces a monotonically increasing local time t_1, and uses it to install a new version $\nu{=}(k_1, v_1, (t_1, r_1), ctx)$, as shown in Fig. 6a. Note that the dependency set of ν is the cl context ($ctx{=}\emptyset$). Replica r_1 then sends the time-stamp (t_1, r_1) back to cl_1, and cl_1 in turn incorporates (k_1, t_1, r_1) in its local context, i.e. cl observes its own write. Finally, r_1 propagates the written version to other replicas *asynchronously* by sending a *synchronisation message* using *causal delivery*: when a replica r' receives a version ν' from another replica r, it waits for all ν' dependencies to arrive at r', and then accepts ν'. As such, the set of versions contained in each replica is closed with respect to the DEP relation. In the example above, when other replicas receive ν from r_1, they can immediately accept ν as $\mathsf{deps}(\nu){=}\emptyset$. Note that replicas may accept new versions from different clients in parallel.

To execute its second multi-read transaction, client cl requests to read from the k_1, k_2 keys by sending the message $\{k_1, k_2\}$ to replica r_1 and awaits a reply. Upon receiving this message, r_1 builds a DEP-*closed snapshot* (a mapping from $\{k_1, k_2\}$ to values) in two phases as follows. First, r_1 *optimistically reads* the most recent versions for k_1 and k_2, *one at a time*. This process may be interleaved with other writes and synchronisation messages. For instance, Fig. 6b depicts a scenario where r_1: (1) first reads $(k_1, v_1, (t_1, r_1), \emptyset)$ for k_1 (highlighted); (2) then receives two synchronisation messages from r_2, containing versions $(k_1, v_1', (t_1, r_2), \emptyset)$ and $(k_2, v_2', (t_2, r_2), \{(k_1, t_1, r_2)\})$; and (3) finally reads $(k_2, v_2', (t_2, r_2), \{(k_1, t_1, r_2)\})$ for k_2 (highlighted). As such, the current snapshot for $\{k_1, k_2\}$ are not DEP-closed: $(k_2, v_2', (t_2, r_2), \{(k_1, t_1, r_2)\})$ depends on a k_1 version with time-stamp (t_1, r_2) which is bigger than (t_1, r_1) for k_1. To remedy this, after the first phase of optimistic reads, r_1 combines (unions) all dependency sets of the versions from the first phase as a *re-fetch set*, and uses it to *re-fetch* the most recent version of each key with the biggest time-stamp from the union of the re-fetch set and the versions from the first phase. For instance, in Fig. 6c, replica r_1 re-fetches the newer version $(k_1, v_1', (t_1, r_2), \emptyset)$ for k_1. Finally, the snapshot obtained after the second phase are sent to the client, and then added to the client context. For their specific setting, Lloyd et al. [35] *informally* argue that the snapshot sent to the client is causally consistent. By contrast, we *verify* the COPS protocol with our general definition of causal consistency.

COPS Verification To prove that COPS satisfies causal consistency, we give an operational semantics for COPS that is faithful to the protocol, which allows fine-graind reads and writes and show that COPS traces can be refined to traces in our semantics using $\mathtt{ET}_{\mathtt{CC}}$ in three steps: (1) every COPS trace can be transferred to a normalised COPS trace, in which multiple reads of a transaction are not interleaved by other transactions; and (2) the normalised COPS trace can be refined to traces in our semantics, in which (3) each step satisfies $\mathtt{ET}_{\mathtt{CC}}$.

The COPS Operational semantics describes transitions over abstract states Θ comprising a set of replicas, a set of client contexts and a program. For instance,

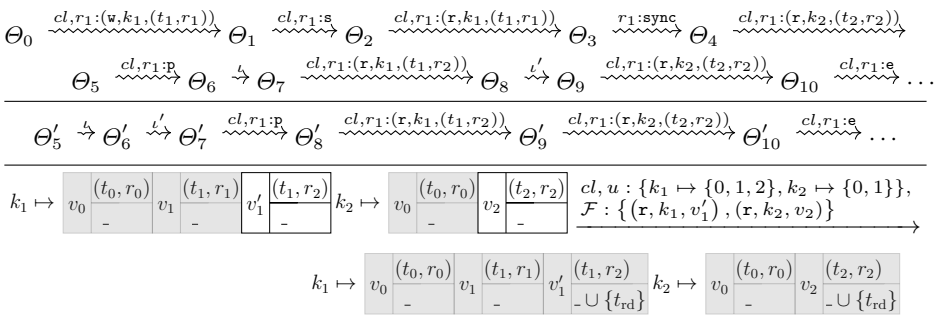


Fig. 7: Top: the COPS trace that produces Figs. 6b and 6c; Middle: the normalised COPS trace; Bottom: the step encoding the multi-read transaction depicted above, with the kv-store encoding of Fig. 6a (left), and the views (highlighted) encoding of the client contexts before and after the transaction.

the COPS trace that produces Figs. 6b and 6c is depicted in Fig. 7 (top), stating that given client cl and replica r_1, (1) cl writes version $(\mathtt{w}, k_1, (t_1, r_1))$ to r_1; (2) cl starts a multi-read transaction ($\mathtt{s}$); (3) cl reads $(\mathtt{r}, k_1, (t_1, r_1))$ from r_1; (4) r_1 receives synchronisation messages ($\mathtt{sync}$); (5) cl enters the second phase of the multi-read transaction ($\mathtt{p}$); (6) cl receives returned versions of the transaction ($\mathtt{e}$).

Recall that a multi-read transaction does not executed atomically in the replica, which is captured by multiple read transitions in the trace. For example, the ι and ι' steps in Fig. 7 (top) interleave the multi-read transaction from cl. Note that the optimistic reads are not observable to the client and thus it suffices to show that the reads from second re-fetch phase are atomic. To show this, we *normalise* the trace as follows. For any multi-read transaction, we move all reads from the re-fetch phase to the right towards the return step $\mathtt{e}$, so that these reads are no longer interleaved by others. An example of a normalised trace is given in Fig. 7 (middle). For any multi-read transaction, the re-fetch phase can only read a version committed before the $\mathtt{p}$ step. For example, in Fig. 7 (top) the multi-read transaction from cl can only read versions in Θ and before. As such, normalising traces does not alter the returned versions of transactions. After normalisation, transactions in the resulting trace can be seen as if executed atomically.

We next show that normalised COPS traces can be refined to traces in our semantics. To do this, we encode the abstract COPS states Θ as configurations in our operational semantics (Fig. 7 (bottom)). We map all COPS replicas to a single kv-store. The writer of a mapped version is uniquely determined by the time-stamp of the corresponding COPS version, while its reader set can be recovered by annotating read-only transactions in the traces such as t_{rd} in Fig. 7(bottom). For example, the COPS state in Fig. 6a can be encoded as the kv-store depicted in Fig. 7 (the state before the transition given in the bottom). Similarly, as the context of a client cl identifies the set of COPS versions that cl sees, we can project COPS client contexts to our client views over kv-stores. For

example, the contexts of cl before and after committing its second multi-read transaction in $\mathtt{P}_{\mathtt{COPS}}$ is encoded as the client view depicted in Fig. 7.

We finally show that every step in the kv-store trace satisfies $\mathtt{ET}_{\mathtt{CC}}$. Note that existing verification techniques [15, 19] require examining the *entire* sequence of operations of a protocol to show that it implements a consistency model. By contrast, we only need to look at how the state evolves after a *single* transaction is executed. In particular, we check the client views over the kv-store. Intuitively, we observe that when a COPS client cl executes a transaction then: (1) the cl context grows, and thus we obtain a more up-to-date view of the associated kv-store; i.e. $\mathsf{vshift}_{\mathtt{MR}}$ holds; (2) the cl context always includes the time-stamp of the versions written by itself, and thus the corresponding client view always includes the versions cl has written; i.e. $\mathsf{vshift}_{\mathtt{RYW}}$ holds; and (3) the cl context is always closed to the relation DEP, which contains the relation $\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}$; i.e. $\mathtt{closed}(\mathcal{K}, u, R_{\mathtt{CC}})$) holds. We have thus demonstrated that COPS satisfies causal consistency; we refer the reader to §H.1 for the full details.

5.2 Application: Invariant Properties of Transactional Libraries

A transactional library provides a set of transactional operations which can be used by its clients to access the underlying kv-store. For instance, the counter library on key k in §2 is $\mathsf{Counter}(k) \triangleq \{\mathsf{inc}(k), \mathsf{read}(k)\}$. Our operational semantics enables us to prove invariant properties of kv-stores, such as: (1) the robustness of the single counter library discussed above against PSI; (2) the robustness of a multi-counter library and the banking library of Alomari et al. [3] against our new proposed model WSI and any stronger model such as SI; and (3) the correctness of a lock paradigm under UA, even though it is not robust.

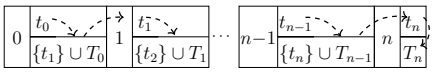
Robustness of a Single Counter against PSI A library L is *robust* against an execution test ET, if for all client programs P of L, the kv-stores $\mathcal{K}$ obtained under ET can also be obtained under SER; that is, if $\mathcal{K} \in \mathsf{CM}(\mathtt{ET})$ then $\mathcal{K} \in \mathsf{CM}(\mathtt{SER})$.

Theorem 1. *A kv-store* $\mathcal{K} \in \mathsf{CM}(\mathtt{SER})$ *iff* $(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{RW}_{\mathcal{K}})^{+} \cap \mathtt{Id} = \emptyset$.

Theorem 1 states that any kv-stores in trace under serialisability must contain no cycle, and vice versa. While previous static-analysis techniques for checking robustness [10, 16, 18, 37] cannot be extended to support client sessions, we give the first robustness proofs that take client sessions into account.

In the single-counter library, $\mathsf{Counter}(k)$, a client reads from k by calling $\mathsf{read}(k)$, and writes to k by calling $\mathsf{inc}(k)$ that first reads the value of k and subsequently increments it by one. As PSI enforces write-conflict freedom (UA), we know that if a transaction t updates k (by calling $\mathsf{inc}(k)$) and writes version ν to k, then it must have read the version of k immediately preceding ν: $\forall t, i > 0. t{=}\mathsf{w}(\mathcal{K}(k,i)) \Rightarrow t \in \mathsf{rs}(\mathcal{K}(k,i{-}1))$. Moreover, as PSI enforces monotonic reads (MR), the order in which clients observe the versions of k (by calling $\mathsf{read}(k)$) is consistent with the order of versions in $\mathcal{K}(k)$. As such, the invariant depicted as below always holds, where $\{t_i\}_{i=1}^{n}$ and $\bigcup_{i=0}^{n} T_i$ denote disjoint sets of transactions calling $\mathsf{inc}(k)$ and $\mathsf{read}(k)$, respectively:

$$(0, t_0, T_0 \cup \{t_1\}) :: (1, t_1, T_1 \cup \{t_2\}) :: \cdots$$
$$:: (n{-}1, t_{n-1}, T_{n-1} \cup \{t_n\}) :: (n, t_n, T_n)$$



We define the $\dashrightarrow$ relation depicted above by extending $\mathsf{SO} \cup \{(t_i, t_j) \mid t_j \in T_i \vee (t_i \in T_i \wedge j{=}i{+}1)\}$ to a strict total order (i.e. an irreflexive and transitive relation). Note that $\dashrightarrow$ contains $\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{RW}_{\mathcal{K}}$ and thus $(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{RW}_{\mathcal{K}})^+$ is irreflexive; i.e. $\mathsf{Counter}(k)$ is robust against PSI.

A multi-counter library on a set of keys K is $\mathsf{Counters}(K) \triangleq \bigcup_{k \in K} \mathsf{Counter}(k)$. Recall from § 2 that unlike in SER and SI, under PSI clients can observe the increments on different keys in different orders (see Fig. 5g). As such, the multi-counter library is not robust against PSI. The following result shows that the multi-counter library is robust against WSI and any stronger model such as SI.

Robustness Conditions against WSI Many libraries [8, 10, 3] yield kv-stores that have a particular pattern that guarantees robustness against WSI:

Definition 11 (WSI-safe). *A kv-store $\mathcal{K}$ is* WSI-safe*, if it is reachable by executing a program* P *under* WSI*,* i.e. $\Gamma_0, \mathtt{P} \rightarrow_{\mathtt{ET}_{\mathtt{WSI}}} (\mathcal{K}, \_), \_$*, and for all t, k, i:*

$$t \neq \mathsf{w}(\mathcal{K}(k,i)) \Rightarrow \forall k', j.\ t \neq \mathsf{w}\big(\mathcal{K}(k',j)\big), \tag{5.1}$$

$$t \neq t_0 \wedge t = \mathsf{w}(\mathcal{K}(k,i)) \Rightarrow \exists j.\ t \in \mathsf{rs}(\mathcal{K}(k,j)), \tag{5.2}$$

$$t \neq t_0 \wedge t = \mathsf{w}(\mathcal{K}(k,i)) \wedge \exists k', j.\ t \in \mathsf{rs}\big(\mathcal{K}(k',j)\big) \Rightarrow t = \mathsf{w}\big(\mathcal{K}(k',j)\big). \tag{5.3}$$

This definition states that a kv-store $\mathcal{K}$ is WSI-safe if for each transaction t: (1) if t does not write to k, then t must be a read-only transaction (5.1); (2) if t writes to k, then it must also read from it (5.2), a property known as *no-blind writes*; and (3) if t writes to k, then it must also write to all keys it reads (5.3). It is straightforward to see that the version j read by t in (5.2) must be written immediately before the version i written by t, i.e. $i{=}j+1$.

Theorem 2 (WSI robustness). *Any* WSI*-safe kv-store $\mathcal{K}$ is robust against* WSI*.*

From Theorem 1 it suffices to prove that $(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{RW}_{\mathcal{K}})^+$ is irreflexive. We proceed by contradiction and assume that there exists t_1 such that $t_1 \xrightarrow{(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{RW}_{\mathcal{K}})^+} t_1$. As $\mathcal{K}$ is reachable under WSI and thus CC, we have:

$$t_1 \xrightarrow{R^*} t_2 \xrightarrow{\mathsf{RW}} t_3 \xrightarrow{R^*} \cdots \xrightarrow{R^*} t_{n-2} \xrightarrow{\mathsf{RW}} t_{n-1} \xrightarrow{R^*} t_n = t_1$$

where $R \triangleq \mathsf{WR} \cup \mathsf{SO} \cup \mathsf{WW}$. From (5.2)(5.3) we know that an RW edge starting from a writing transaction can be replaced by a WW edge. Moreover, WW edges can be replaced by WR^* since $\mathcal{K}$ is reachable under UA. We thus have:

$$t_1 \xrightarrow{R'^*} t'_2 \xrightarrow{\mathsf{RW}} t'_3 \xrightarrow{R'^+} \cdots \xrightarrow{R'^+} t'_{m-2} \xrightarrow{\mathsf{RW}} t'_{m-1} \xrightarrow{R'^*} t'_m = t_n = t_1$$

where $R' \triangleq \mathsf{WR} \cup \mathsf{SO}$. That is, $t_1 \xrightarrow{(R';\mathsf{RW}^?)^*} t_1$. This leads to a contradiction as $R';\mathsf{RW}^? \subseteq R_{\mathtt{CP}}$ and WSI requires views to be closed under $R_{\mathtt{CP}}$ (Fig. 4).

Robustness of a Banking Library against WSI Using Theorem 2, we are able to prove the robustness of the banking library in Alomari et al. [3] against WSI. The banking example is based on relational databases and has three tables: *account*, *saving* and *checking*. The account table maps customer names to customer IDs ($\mathtt{Account}(\underline{\mathtt{Name}}, \mathtt{CID})$). The saving table maps customer IDs to their saving balances ($\mathtt{Saving}(\underline{\mathtt{CID}}, \mathtt{Balance})$), and the checking table maps customer IDs to their checking balances ($\mathtt{Checking}(\underline{\mathtt{CID}}, \mathtt{Balance})$).

For simplicity, we encode the saving and checking tables as a kv-store, and forgo the account table as it is an immutable lookup table. We model a customer ID as an integer $n \in \mathbb{N}$, and assume that balances are integer values. We then define the key associated with customer n in the checking table as $n_c \triangleq 2n$, and define the key associated with n in the saving table as $n_s \triangleq 2n{+}1$. That is, $\textsc{Keys} \triangleq \bigcup_{n\in\mathbb{N}} \{n_c, n_s\}$. Moreover, if n identifies a customer, i.e. $(\_, n) \in \texttt{Account}(\underline{\texttt{Name}}, \texttt{CID})$, then $(n, \mathsf{val}(\mathcal{K}(n_s, |\mathcal{K}(n_s)| - 1))) \in \texttt{Saving}(\underline{\texttt{CID}}, \texttt{Balance})$ and $(n, \mathsf{val}(\mathcal{K}(n_c, |\mathcal{K}(n_c)| - 1))) \in \texttt{Checking}(\underline{\texttt{CID}}, \texttt{Balance})$.

The banking library provides five transactional operations. The $\texttt{balance}(\texttt{n})$ operation returns the total balance of customer $\texttt{n}$ in $\texttt{ret}$. The $\texttt{depositCheck}(\texttt{n}, \texttt{v})$ operation deposits $\texttt{v}$ to the checking account of customer $\texttt{n}$ when $\texttt{v}$ is non-negative, otherwise the checking account remains unchanged.

$$\texttt{balance}(\texttt{n}) \triangleq [\texttt{x} := [\texttt{n}_c]\,;\ \texttt{y} := [\texttt{n}_s]\,;\ \texttt{ret} := \texttt{x} + \texttt{y}]$$

$$\texttt{depositCheck}(\texttt{n}, \texttt{v}) \triangleq [\texttt{if}\ (\texttt{v} \geq 0)\{\ \texttt{x} := [\texttt{n}_c]\,;\ [\texttt{n}_c] := \texttt{x} + \texttt{v};\ \}]$$

$$\texttt{transactSaving}(\texttt{n}, \texttt{v}) \triangleq [\texttt{x} := [\texttt{n}_s]\,;\ \texttt{if}\ (\texttt{v} + \texttt{x} \geq 0)\{\ [\texttt{n}_s] := \texttt{x} + \texttt{v};\ \}]$$

$$\texttt{amalgamate}(\texttt{n}, \texttt{n}') \triangleq \begin{bmatrix} \texttt{x} := [\texttt{n}_s]\,;\ \texttt{y} := [\texttt{n}_c]\,;\ \texttt{z} := [\texttt{n}'_c]\,; \\ [\texttt{n}_s] := 0;\ [\texttt{n}_c] := 0;\ [\texttt{n}'_c] := \texttt{x} + \texttt{y} + \texttt{z}; \end{bmatrix}$$

$$\texttt{writeCheck}(\texttt{n}, \texttt{v}) \triangleq \begin{bmatrix} \texttt{x} := [\texttt{n}_s]\,;\ \texttt{y} := [\texttt{n}_c]\,; \\ \texttt{if}\ (\texttt{x} + \texttt{y} < \texttt{v})\{\ [\texttt{n}_c] := \texttt{y} - \texttt{v} - 1;\ \} \\ \texttt{else}\{\ [\texttt{n}_c] := \texttt{y} - \texttt{v};\ \}\ \ [\texttt{n}_s] := \texttt{x}; \end{bmatrix}$$

The $\texttt{balance}(\texttt{n})$ operation returns the total balance of customer $\texttt{n}$ in $\texttt{ret}$. The $\texttt{depositChecking}(\texttt{n}, \texttt{v})$ operation deposits $\texttt{v}$ to the checking account of customer $\texttt{n}$ when $\texttt{v}$ is non-negative, otherwise the checking account remains unchanged. When $\texttt{v} \geq 0$, $\texttt{transactSaving}(\texttt{n}, \texttt{v})$ deposits $\texttt{v}$ to the saving account of $\texttt{n}$. When $\texttt{v} < 0$, $\texttt{transactSaving}(\texttt{n}, \texttt{v})$ withdraws $\texttt{v}$ from the saving account of $\texttt{n}$ only if the resulting balance is non-negative, otherwise the saving account remains unchanged. The $\texttt{amalgamate}(\texttt{n}, \texttt{n}')$ operation moves the combined checking and saving funds of costumer $\texttt{n}$ to the checking account of customer $\texttt{n}'$. Lastly, $\texttt{writeCheck}(\texttt{n}, \texttt{v})$ cashes a cheque of customer $\texttt{n}$ in the amount $\texttt{v}$ by deducting $\texttt{v}$ from its checking account. If $\texttt{n}$ does not hold sufficient funds (i.e. the combined checking and saving balance is less than $\texttt{v}$), customer $\texttt{n}$ is penalised by deducting one additional pound. Alomari et al. [3] argue that to make the banking library robust against SI, the $\texttt{writeCheck}(\texttt{n}, \texttt{v})$ operation must be strengthened by writing back the balance to the saving account (via $[\texttt{n}_s] := \texttt{x}$), even though the saving balance is unchanged. The banking library is more complex than the multi-counter library. Nevertheless, all banking transactions are either read-only or satisfy the strictly-no-blind writes property; i.e. the banking library is WSI-safe. As such, we can prove its robustness against WSI and hence SI.

Lock Invariant The distributed lock library provides $\texttt{lock}\,(\texttt{k})$, $\texttt{trylock}\,(\texttt{k})$ and $\texttt{unlock}\,(\texttt{k})$ operations on the key k:

$$\texttt{trylock}\,(\texttt{k}) \triangleq [\texttt{x} := [k]\,;\ \texttt{if}\ (\texttt{x}{=}0)\{\ [k] := \texttt{ClientID};\ \texttt{m} := \texttt{true}\ \}\texttt{else}\{\ \texttt{m} := \texttt{false}\ \}]$$

$$\texttt{lock}\,(\texttt{k}) \triangleq \texttt{m} := \texttt{false};\ \texttt{while}(\texttt{m}{=}\texttt{false})\{\ \texttt{trylock}\,(\texttt{k})\ \} \qquad \texttt{unlock}\,(\texttt{k}) \triangleq [[k] := 0]$$

The $\texttt{trylock}$ operation reads the k vale. If it is zero, i.e. the lock is available, the operation sets it to the client ID and returns $\texttt{true}$; otherwise it leaves it

unchanged and returns `false`. The `lock` operation calls `trylock` until it successfully acquires the lock. The `unlock` operation simply set the value to zero.

Consider the program $\mathtt{P}_{\mathtt{LK}}$ where clients cl and cl' compete the lock k:

$$\mathtt{P}_{\mathtt{LK}} \triangleq (cl : (\mathtt{lock}\,(\mathtt{k});\ ...;\ \mathtt{unlock}\,(\mathtt{k}))^* \parallel cl' : (\mathtt{lock}\,(\mathtt{k});\ ...;\ \mathtt{unlock}\,(\mathtt{k}))^*)$$

The locking paradigm in $\mathtt{P}_{\mathtt{LK}}$ is correct, in that only one client can hold the lock at a time, when executed under serialisability. Since all the operations are trivially `WSI`-safe, $\mathtt{P}_{\mathtt{LK}}$ is robust and hence correct under `WSI` and any stronger model such as `SI`. However, $\mathtt{P}_{\mathtt{LK}}$ is not robust under `UA`, because the `lock` operation might read any old value of key k until it reads the up-to-date value of k and acquires the lock. Nevertheless, we show that $\mathtt{P}_{\mathtt{LK}}$ is still correct under `UA`. We capture thus by the following invariant: for all i, if $i > 0$, then:

$$\mathsf{val}(\mathcal{K}\,(k,i)) \neq 0 \Leftrightarrow \mathsf{val}(\mathcal{K}\,(k,i-1)) = 0 \tag{5.4}$$

$$\mathsf{val}(\mathcal{K}\,(k,i)) = 0 \Rightarrow \mathsf{w}(\mathcal{K}\,(k,i)) = \mathsf{w}(\mathcal{K}\,(k,i-1)) \tag{5.5}$$

It is straightforward to show that under `UA`, only one client can hold the key (5.4), and the same client releases the lock (5.5).

6 Correct Definitions of Consistency Models

We demonstrate how our kv-stores and execution tests relate to existing declarative semantics for specifying consistency models, based on abstract executions [15]. We prove our definitions of consistency models using execution tests are *equivalent* to the definitions on abstract executions. We give an overview of our results here, and refer the reader to appendix B and C for more details.

Abstract executions [13, 15] are a declarative formalism for defining consistency models. An abstract execution graph is a directed graph with its nodes representing transactions (with each node labelled with a transaction identifier and a set of read/write operations), and its edges representing certain relations between transactions. Each edge is labelled by either the *visibility* (VIS) or *arbitration* (AR) relation. Visibility is an irreflexive order on transactions such that $(t_1, t_2) \in \mathsf{VIS}$ denotes that the effects (updates) of t_1 are visible to t_2. Arbitration is a strict total order on transactions such that $(t_1, t_2) \in \mathsf{AR}$ denotes that the updates performed by t_2 are newer than those of t_1. Moreover, AR contains VIS ($\mathsf{VIS} \subseteq \mathsf{AR}$) and agrees with the session order. Lastly, abstract executions observe the *last-write-wins* policy: a transaction reading k always fetches the latest visible write on k. Consistency models are defined by visibility axioms $\mathcal{A}$ that impose certain conditions on VIS and hence the shape of the graphs.

We show that there is a correspondence between kv-stores and abstract executions. To do this, we (1) establish a bijection between kv-stores and dependency graphs (§B), another well-known graph-based declarative formalism; and then (2) following [18], we show that we can always *extract* an abstract execution from a dependency graph and therefore from a kv-store by (1). However, this correspondence is not one-to-one as multiple AR relations may produce equivalent kv-stores. As such, our correspondence result is not enough to prove that our definitions using `ET` are equivalent to those using $\mathcal{A}$. This is because `ET` constrains each transition step while $\mathcal{A}$ constrains the final graph shape. We

thus propose an alternative operational semantics on abstract executions, where each transition extends the graph with a new transaction node and its visibility edges, provided that the new graph satisfies $\mathcal{A}$.

We thus show that a consistency model defined using an execution test `ET` is *equivalent* (i.e. *sound* and *complete*) to the definition using visibility axiom $\mathcal{A}$, if for every program there is a correspondence between traces of kv-stores under `ET` and traces of abstract executions under $\mathcal{A}$. Instead of directly working on traces, we introduce soundness and completeness constructors (§E) that lift certain conditions between `ET` and $\mathcal{A}$ to the level of traces. In §F, we show that all our definitions of consistency models in Fig. 4 are *equivalent* to existing axiomatic definitions using abstract executions.

7 Conclusions and Related Work

We have introduced a simple interleaving semantics for atomic transactions, based on a global, centralised kv-stores and partial client views. It is expressive enough to capture the anomalous behaviour of many weak consistency models. We have demonstrated that our semantics can be used to both verify protocols of distributed databases and prove the invariant properties of client programs.

We have defined a large variety of consistency models for kv-stores based on execution tests, and have shown these models to be equivalent to well-known declarative consistency models for dependency graphs and abstract executions. We do not know of an appropriate consistency model that we cannot express with our semantics, bearing in mind the constraints that our transactions satisfy snapshot property and the last-write-wins policy. We have identified a new consistency model, *weak snapshot isolation*, which lies between `PSI` and `SI` and inherits many of the good properties of `SI`. We have shown that examples are robust against `WSI`. We would need to provide an implementation of this model to justify it in full in the future. We have proved the correctness of two real-world protocols employed by distributed databases: COPS [35], a replicated database that satisfies causal consistency; and Clock-SI [24], a partitioned database that satisfies snapshot isolation. We have also demonstrated the usefulness of our framework for proving invariant properties: the robustness properties of the counters and banking libraries against different consistency models; and the correctness (despite not robust) of the lock paradigm under `UA`.

In future, we aim to extend our framework to handle other weak consistency models. For example, we believe that, by introducing promises in the style of [29], we can capture consistency models such as *Read Committed*. We also plan to validate further the usefulness of our framework by: verifying other well-known protocols of distributed databases [36, 46, 32, 5]; exploring robustness results for OLTP workloads such as TPC-C [47] and RUBiS [41]; and exploring other program analysis techniques such as transaction chopping [43, 17], invariant checking [26, 49] and program logics [28]. We plan to develop tools to generate litmus tests for implementations and to analyse the invariant of client programs.

Related Work In the introduction, we highlight some general operational semantics for distributed transactional databases. We discuss these semantics in

more detail here. There are graph-based, state-based and log-based operational semantics. We then give some additional related work on program analysis.

Nagar and Jagannathan [37] propose an operational semantics for weak consistency based on abstract executions. Their semantics is parametric in the declarative definition of a consistency model. They introduce a tool for checking the robustness of transactional libraries. They focus on consistency models with snapshot property, but confusingly allow the interleaving of fine-grained operations between transactions. This results in an unnecessary explosion of the space of traces obtained by the program. In our semantics, the interleaving is between transactions. Doherty et al. [23] develop an operational semantics for release-acquire fragment of C11 memory model, a variant of causal consistency. Their semantics is based on a variant of dependency graph where nodes and edges are tailored for C11 operations. They introduce per-thread observations that are compatible for executing next operations; this is similar to our views and execution tests. We believe we can model release-acquire fragment of C11.

Crooks et al. [19] propose a state-based formal framework for weak consistency models that employs concepts similar to execution tests and views, called commit tests and read states respectively. They prove that consistency models previously thought to be different are in fact equivalent in their semantics. They capture a wide range of consistency models including read committed which we cannot do. In their semantics, one-step trace reduction is determined by the whole previous history of the trace. In contrast, our reduction step only depends on the current configuration (kv-store and view). They do not consider program analysis. Their notion of commit tests and read states requires the knowledge of information that is not known to clients of the system, i.e. the total order of system changes that happened in the database prior to committing a transaction. For this reason, we believe that their framework is not suitable for the development of techniques for analysing client programs. Kaki et al. [28] propose an operational semantics of SQL transactional programs under the consistency models given by the standard ANSI/SQL isolation levels [9]. In their framework, transactions work on a local copy of the global state of the system, and the local effects of a transaction are committed to the system state when it terminates. Because state changes are made immediately available to all clients of a system, this model is not suitable to capture weak consistency models such as `PSI` or `CC`. They introduce a program logic and prototype verification tool for reasoning about client programs. However, their definitions of consistency models are not validated against well-known formal definitions.

Koskinen et al. [31] propose two log-based abstract operational semantics, pessimistic one and optimistic one, and Koskinen and Parkinson [30] propose the push/pull semantics, both of which are used to verify implementations of serilisability. The machine state of the push/pull semantics consists of a centralised global log and partial client local logs. This is similar to our kv-stores and views. Both semantics can model key-value stores and also other ADTs. However, their model is thought with serialisability in mind. There is no evidence that they could be easily extended to tackle weaker consistency models.

Several other works have focused on program analysis for transactional systems. Dias et al. [21] developed a separation logic for the robustness of applications against `SI`. Fekete et al. [25] derived a static analysis check for `SI` based on dependency graph. Bernardi and Gotsman [10] developed a static analysis check for several consistency models with snapshot property. Beillahi et al. [8] developed a tool based on Lipton's reduction [33] for checking robustness against `SI`. Cerone et al. [18] investigated the relationship between abstract executions and dependency graphs from an algebraic perspective, and applied it to infer robustness checks for several consistency models.

Bibliography

[1] Adya, A.: Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA (1999)

[2] Adya, A., Liskov, B., O'Neil, P.E.: Generalized isolation level definitions. In: ICDE (2000)

[3] Alomari, M., Cahill, M., Fekete, A., Rohm, U.: The cost of serializability on platforms that use snapshot isolation. In: 2008 IEEE 24th International Conference on Data Engineering, pp. 576–585 (April 2008), ISSN 1063-6382, https://doi.org/10.1109/ICDE.2008.4497466

[4] Ardekani, M.S., Sutra, P., Shapiro, M.: G-DUR: A Middleware for Assembling, Analyzing, and Improving Transactional Protocols. In: Proceedings of the 15th International Middleware Conference, pp. 13–24, Middleware'14, ACM, New York, NY, USA (2014), ISBN 978-1-4503-2785-5, https://doi.org/10.1145/2663165.2663336

[5] Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Scalable Atomic Visibility with RAMP Transactions. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 27–38 (2014)

[6] Balakrishnan, M., Malkhi, D., Davis, J.D., Prabhakaran, V., Wei, M., Wobber, T.: Corfu: A distributed shared log. ACM Trans. Comput. Syst. **31**(4), 10:1–10:24 (December 2013), ISSN 0734-2071, https://doi.org/10.1145/2535930, URL `http://doi.acm.org/10.1145/2535930`

[7] Balakrishnan, M., Malkhi, D., Wobber, T., Wu, M., Prabhakaran, V., Wei, M., Davis, J.D., Rao, S., Zou, T., Zuck, A.: Tango: Distributed data structures over a shared log. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 325–340, SOSP '13, ACM, New York, NY, USA (2013), ISBN 978-1-4503-2388-8, https://doi.org/10.1145/2517349.2522732, URL `http://doi.acm.org/10.1145/2517349.2522732`

[8] Beillahi, S.M., Bouajjani, A., Enea, C.: Checking robustness against snapshot isolation. CoRR **abs/1905.08406** (2019), URL `http://arxiv.org/abs/1905.08406`

[9] 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, pp. 1–10, SIGMOD'95, ACM (1995), https://doi.org/10.1145/223784.223785

[10] Bernardi, G., Gotsman, A.: Robustness against Consistency Models with Atomic Visibility. In: Proceedings of the 27th International Conference on Concurrency Theory, pp. 7:1–7:15 (2016), https://doi.org/10.4230/LIPIcs.CONCUR.2016.7

[11] 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 **23**(6), 987–1011 (December 2014), ISSN 1066-8888, https://doi.org/10.1007/s00778-014-0359-9

[12] Burckhardt, S.: Principles of eventual consistency. Found. Trends Program. Lang. **1**(1-2), 1–150 (October 2014), ISSN 2325-1107, https://doi.org/10.1561/2500000011, URL `http://dx.doi.org/10.1561/2500000011`

[13] Burckhardt, S., Fahndrich, M., Leijen, D., Sagiv, M.: Eventually Consistent Transactions. In: Proceedings of the 21$^{\text{nd}}$ European Symposium on Programming, Springer (March 2012)

[14] Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated Data Types: Specification, Verification, Optimality. In: Proceedings of the 41$^{\text{st}}$ Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 271–284, POPL'14, ACM (2014)

[15] Cerone, A., Bernardi, G., Gotsman, A.: A Framework for Transactional Consistency Models with Atomic Visibility. In: Aceto, L., de Frutos-Escrig, D. (eds.) Proceedings of the 26$^{\text{th}}$ International Conference on Concurrency Theory, Leibniz International Proceedings in Informatics (LIPIcs), vol. 42, pp. 58–71, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2015), https://doi.org/10.4230/LIPIcs.CONCUR.2015.58

[16] Cerone, A., Gotsman, A.: Analysing Snapshot Isolation. In: Proceedings of the 2016 ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pp. 55–64, PODC'16, ACM (2016), https://doi.org/10.1145/2933057.2933096

[17] Cerone, A., Gotsman, A., Yang, H.: Transaction Chopping for Parallel Snapshot Isolation. In: Proceedings of the 29$^{\text{th}}$ International Symposium on Distributed Computing, pp. 388–404 (2015)

[18] Cerone, A., Gotsman, A., Yang, H.: Algebraic Laws for Weak Consistency. In: Meyer, R., Nestmann, U. (eds.) Proceedings of the 27$^{\text{th}}$ International Conference on Concurrency Theory, Leibniz International Proceedings in Informatics (LIPIcs), vol. 85, pp. 26:1–26:18, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2017), https://doi.org/10.4230/LIPIcs.CONCUR.2017.26

[19] 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, pp. 73–82, PODC'17, ACM, New York, NY, USA (2017), ISBN 978-1-4503-4992-5, https://doi.org/10.1145/3087801.3087802

[20] da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: A Logic for Time and Data Abstraction. In: Jones, R.E. (ed.) Proceedings of the 28$^{\text{th}}$ European Conference on Object-Oriented Programming, Lecture Notes in Computer Science, vol. 8586, pp. 207–231, Springer (July 2014)

[21] Dias, R.J., Distefano, D., Seco, J.C., Lourenço, J.M.: Verification of snapshot isolation in transactional memory java programs. In: Noble, J. (ed.)

ECOOP 2012 – Object-Oriented Programming, pp. 640–664, Springer Berlin Heidelberg, Berlin, Heidelberg (2012), ISBN 978-3-642-31057-7

[22] Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent Abstract Predicates. In: Proceedings of the 24th European Conference on Object-Oriented Programming, pp. 504–528, ECOOP'10, Springer-Verlag (2010)

[23] Doherty, S., Dongol, B., Wehrheim, H., Derrick, J.: Verifying c11 programs operationally. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, pp. 355–365, PPoPP '19, ACM, New York, NY, USA (2019), ISBN 978-1-4503-6225-2, https://doi.org/10.1145/3293883.3295702, URL http://doi.acm.org/10.1145/3293883.3295702

[24] 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), pp. 173–184, SRDS'13, IEEE Computer Society, Washington, DC, USA (2013), ISBN 978-0-7695-5115-9, https://doi.org/10.1109/SRDS.2013.26, URL https://doi.org/10.1109/SRDS.2013.26

[25] Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., Shasha, D.: Making Snapshot Isolation Serializable. ACM Transactions on Database Systems **30**(2), 492–528 (June 2005), https://doi.org/10.1145/1071610.1071615

[26] Gotsman, A., Yang, H., Ferreira, C., Najafzadeh, M., Shapiro, M.: 'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems. In: Bodik, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 371–384, ACM (2016)

[27] Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 637–650, POPL'15, ACM (2015), https://doi.org/10.1145/2676726.2676980

[28] Kaki, G., Nagar, K., Najafzadeh, M., Jagannathan, S.: Alone Together: Compositional Reasoning and Inference for Weak Isolation. Proceedings of the ACM on Programming Languages **2**(POPL), 27:1–27:34 (December 2017), ISSN 2475-1421, https://doi.org/10.1145/3158115, URL http://doi.acm.org/10.1145/3158115

[29] Kang, J., Hur, C.K., Lahav, O., Vafeiadis, V., Dreyer, D.: A Promising Semantics for Relaxed-memory Concurrency. In: Proceedings of the 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 175–189, POPL'17, ACM, New York, NY, USA (2017), ISBN 978-1-4503-4660-3, https://doi.org/10.1145/3009837.3009850, URL http://doi.acm.org/10.1145/3009837.3009850

[30] Koskinen, E., Parkinson, M.: The Push/Pull Model of Transactions. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 186–195, PLDI'15, ACM (2015)

[31] Koskinen, E., Parkinson, M., Herlihy, M.: Coarse-grained transactions. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 19–30, POPL '10, ACM, New York, NY, USA (2010), ISBN 978-1-60558-479-9, https://doi.org/10.1145/1706299.1706304, URL http://doi.acm.org/10.1145/1706299.1706304

[32] Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., Rodrigues, R.: Making geo-replicated systems fast as possible, consistent when necessary. In: Proceedings of the 10th Symposium on Operating Systems Design and Implementation, pp. 265–278 (2012)

[33] Lipton, R.J.: Reduction: A method of proving properties of parallel programs. Commun. ACM **18**(12), 717–721 (December 1975), ISSN 0001-0782, https://doi.org/10.1145/361227.361234, URL http://doi.acm.org/10.1145/361227.361234

[34] Liu, S., Ölveczky, P.C., Santhanam, K., Wang, Q., Gupta, I., Meseguer, J.: ROLA: A New Distributed Transaction Protocol and Its Formal Analysis. In: Russo, A., Schürr, A. (eds.) Fundamental Approaches to Software Engineering, pp. 77–93, Springer, Cham (2018)

[35] Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles, pp. 401–416, SOSP'11, ACM (2011)

[36] Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Stronger Semantics for Low-Latency Geo-Replicated Storage. In: Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation, pp. 313–328, USENIX, Lombard, IL (2013), ISBN 978-1-931971-00-3, URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/lloyd

[37] Nagar, K., Jagannathan, S.: Automated Detection of Serializability Violations Under Weak Consistency. In: Proceedings of the 29th International Conference on Concurrency Theory, pp. 41:1–41:18 (2018), https://doi.org/10.4230/LIPIcs.CONCUR.2018.41, URL https://doi.org/10.4230/LIPIcs.CONCUR.2018.41

[38] Nanevski, A., Ley-wild, Y., Sergey, I., Delbianco, G.A.: Communicating State Transition Systems for fine-grained concurrent resources, pp. 290–310. Lecture Notes in Computer Science, springer-verlag (2014)

[39] Petersen, K., Spreitzer, M.J., Terry, D.B., Theimer, M.M., Demers, A.J.: Flexible update propagation for weakly consistent replication. In: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, pp. 288–301, SOSP '97, ACM, New York, NY, USA (1997), ISBN 0-89791-916-5, https://doi.org/10.1145/268998.266711, URL http://doi.acm.org/10.1145/268998.266711

[40] Raad, A., Lahav, O., Vafeiadis, V.: On Parallel Snapshot Isolation and Release/Acquire Consistency. In: Ahmed, A. (ed.) Proceedings of the 27th European Symposium on Programming, pp. 940–967, Lecture Notes in Computer Science, Cham (2018), ISBN 978-3-319-89884-1

[41] RUBIS: The RUBiS benchmark. https://rubis.ow2.org/index.html (2008)
[42] Saeida Ardekani, M., Sutra, P., Shapiro, M.: Non-monotonic Snapshot Isolation: Scalable and Strong Consistency for Geo-replicated Transactional Systems. In: Proceedings of the 32nd Leibniz International Proceedings in Informatics (LIPIcs), pp. 163–172 (2013)
[43] Shasha, D., Llirbat, F., Simon, E., Valduriez, P.: Transaction Chopping: Algorithms and Performance Studies. ACM Transactions on Database Systems **20**(3), 325–363 (September 1995), ISSN 0362-5915, https://doi.org/10.1145/211414.211427, URL http://doi.acm.org/10.1145/211414.211427
[44] Sivaramakrishnan, K., Kaki, G., Jagannathan, S.: Declarative programming over eventually consistent data stores. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 413–424, PLDI '15, ACM, New York, NY, USA (2015), ISBN 978-1-4503-3468-6, https://doi.org/10.1145/2737924.2737981, URL http://doi.acm.org/10.1145/2737924.2737981
[45] Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional Storage for Geo-replicated Systems. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles, pp. 385–400, SOSP'11, ACM, New York, NY, USA (2011), ISBN 978-1-4503-0977-6, https://doi.org/10.1145/2043556.2043592, URL http://doi.acm.org/10.1145/2043556.2043592
[46] Spirovska, K., Didona, D., Zwaenepoel, W.: Wren: Nonblocking Reads in a Partitioned Transactional Causally Consistent Data Store. In: Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 1–12, DSN'18 (2018), https://doi.org/10.1109/DSN.2018.00014, URL https://doi.org/10.1109/DSN.2018.00014
[47] TPCC: The TPC-C benchmark. http://www.tpc.org/tpcc/ (1992)
[48] Vogels, W.: Eventually Consistent. Communications of the ACM **52**(1), 40–44 (January 2009)
[49] Zeller, P.: Testing properties of weakly consistent programs with repliss. In: Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data, pp. 3:1–3:5, PaPoC'17, ACM, New York, NY, USA (2017), ISBN 978-1-4503-4933-8, https://doi.org/10.1145/3064889.3064893, URL http://doi.acm.org/10.1145/3064889.3064893

A Operational Semantics on KV-Stores

Definition 12 (Multi-version Key-value Stores). *Assume a countably infinite set of* keys $\textsc{Keys} \ni k$*, and a countably infinite set of* values $\textsc{Val} \ni v$*, including an* initialisation value v_0*. The set of* versions, $\textsc{Version} \ni \nu$*, is:* $\textsc{Version} \triangleq \textsc{Val} \times \textsc{TransID} \times \mathcal{P}(\textsc{TransID}_0)$*. A* kv-store *is a function* $\mathcal{K} : \textsc{Keys} \to \mathsf{List}(\textsc{Version})$*, where* $\mathsf{List}(\textsc{Version}) \ni \mathcal{V}$ *is the set of lists of versions* Version*. Well-formed key-values store satisfy:*

$$\forall k, i, j.\ \mathsf{rs}(\mathcal{K}(k,i)) \cap \mathsf{rs}(\mathcal{K}(k,j)) \neq \emptyset \vee \mathsf{w}(\mathcal{K}(k,i)) = \mathsf{w}(\mathcal{K}(k,j)) \Rightarrow i = j \tag{1.1}$$

$$\forall k.\ \mathcal{K}(k,0) = (v_0, t_0, \_) \tag{1.2}$$

$$\forall k, cl, i, j, n, m.\ t^n_{cl} = \mathsf{w}(\mathcal{K}(k,i)) \wedge t^m_{cl} \in \{\mathsf{w}(\mathcal{K}(k,j))\} \cup \mathsf{rs}(\mathcal{K}(k,i)) \Rightarrow n < m \tag{1.3}$$

The full semantics is in Fig. 8 and the full definition of consistency models is in Fig. 9.

Definition 13 (ET-reduction). *An* $\texttt{ET}$-reduction, $(\mathcal{K},\mathcal{U}) \xrightarrow{(cl,\alpha)}_{\texttt{ET}} (\mathcal{K}',\mathcal{U}')$*, is defined by:*

(1) either $\alpha = \varepsilon$*,* $\mathcal{K}' = \mathcal{K}$ *and* $\mathcal{U}' = \mathcal{U}[cl \mapsto u]$ *for some* u *such that* $\mathcal{U}(cl) \sqsubseteq u$*; or*

(2) $\alpha = \mathcal{F}$ *for some* $\mathcal{F}$*, and* $\texttt{ET} \vdash (\mathcal{K}, u) \triangleright \mathcal{F} : (\mathcal{K}', u')$*, where* $\mathcal{K}' = \mathsf{update}(\mathcal{K}, u, \mathcal{F}, t)$ *for some* $t \in \mathsf{nextTid}(cl, \mathcal{K})$*,* $\mathcal{U}' = \mathcal{U}[cl \mapsto u']$*.*

A finite sequence of $\texttt{ET}$*-reductions starting in an initial configuration* Γ_0 *is called an* $\texttt{ET}$-trace*.*

Each $\texttt{ET}$-trace starting with an initial configuration (Def. 3) terminates in a configuration $(\mathcal{K}, \_)$ where $\mathcal{K}$ is obtained as a result of several clients committing transactions under the execution test $\texttt{ET}$. The consistency model induced by $\texttt{ET}$, written $\mathsf{CM}(\texttt{ET})$, is the set of all such terminal kv-stores.

Note that in the definition of $\texttt{ET}$-traces, the view-shifts and transaction commits are decoupled. This is in contrast to the operational semantics (§ 3, Fig. 3), where view-shifts and transaction commits are combined in a single transition of programs (CAtomicTrans). The reason for this mismatch is best understood when looking at the intended applications. ET-traces are useful for proving that a distributed transactional protocol implements a given consistency model: in this case, it is convenient to separate shifting a view from committing a transaction, as these two steps often take place separately in distributed protocols. The operational semantics is particularly useful for reasoning about transactional programs: in this case, the treatment of the view-shifts and transaction commits as a single transition reduces the number of interleavings in programs. The $\texttt{ET}$-traces and operational semantics are equally expressive as the following theorem states.

Theorem 3. *Let* $[\![\texttt{P}]\!]_{\texttt{ET}}$ *be the set of kv-stores reachable by executing* $\texttt{P}$ *under the execution test* $\texttt{ET}$ *and* $[\![\texttt{ET}]\!]$ *be the the set of kv-stores reachable by* $\texttt{ET}$*-tarces. Then for all* $\texttt{ET}$*,* $[\![\texttt{ET}]\!] = \bigcup_{\texttt{P}} [\![\texttt{P}]\!]_{\texttt{ET}}$*.*

$\rightarrow$: ((Stack $\times$ Snapshot $\times$ Fp)
$\times$ Transactions) $\times$ ((Stack $\times$ Snapshot $\times$ Fp) $\times$ Transactions)

TPrimitive
$$\frac{(s,\sigma) \overset{\mathtt{T}_p}{\rightsquigarrow} (s',\sigma') \qquad o = \mathsf{op}(s,\sigma,\mathtt{T}_p)}{(s,\sigma,\mathcal{F}),\mathtt{T}_p \rightarrow (s',\sigma',\mathcal{F} \lll o),\mathtt{skip}}$$

TChoice
$$\frac{i \in \{1,2\}}{(s,\sigma,\mathcal{F}),\mathtt{T}_1 + \mathtt{T}_2 \rightarrow (s,\sigma,\mathcal{F}),\mathtt{T}_i}$$

TIter
$$\frac{}{(s,\sigma,\mathcal{F}),\mathtt{T}^* \rightarrow (s,\sigma,\mathcal{F}),\mathtt{skip} + (\mathtt{T}\,;\mathtt{T}^*)}$$

TSeqSkip
$$\frac{}{(s,\sigma,\mathcal{F}),\mathtt{skip}\,;\mathtt{T} \rightarrow (s,\sigma,\mathcal{F}),\mathtt{T}}$$

TSeq
$$\frac{(s,\sigma,\mathcal{F}),\mathtt{T}_1 \rightarrow (s',\sigma',\mathcal{F}'),\mathtt{T}_1'}{(s,\sigma,\mathcal{F}),\mathtt{T}_1\,;\mathtt{T}_2 \rightarrow (s',\sigma',\mathcal{F}'),\mathtt{T}_1'\,;\mathtt{T}_2}$$

$\rightarrow$: ClientID $\times$ ((KVS $\times$ Views $\times$ Stack) $\times$ Commands)
$\times$ ET $\times$ Labels $\times$ ((KVS $\times$ Views $\times$ Stack) $\times$ Commands)

CAtomicTrans
$$\frac{\begin{array}{ccc} & u \sqsubseteq u'' & \\ \sigma = \mathsf{snapshot}(\mathcal{K},u'') & (s,\sigma,\emptyset),\mathtt{T} \rightarrow^* (s',\_,\mathcal{F}),\mathtt{skip} & t \in \mathsf{nextTid}(cl,\mathcal{K}) \\ \mathcal{K}' = \mathsf{update}(\mathcal{K},u'',\mathcal{F},t) & \mathsf{can\text{-}commit}_{\mathtt{ET}}(\mathcal{K},u'',\mathcal{F}) & \mathsf{vshift}_{\mathtt{ET}}(\mathcal{K},u'',\mathcal{K}',u') \end{array}}{cl \vdash (\mathcal{K},u,s),[\mathtt{T}] \xrightarrow{(cl,u'',\mathcal{F})}_{\mathtt{ET}} (\mathcal{K}',u',s'),\mathtt{skip}}$$

CPrimitive
$$\frac{s \overset{\mathtt{C}_p}{\rightsquigarrow} s'}{cl \vdash (\mathcal{K},u,s),\mathtt{C}_p \xrightarrow{(cl,\iota)}_{\mathtt{ET}} (\mathcal{K},u,s'),\mathtt{skip}}$$

CChoice
$$\frac{i \in \{1,2\}}{cl \vdash (\mathcal{K},u,s),\mathtt{C}_1 + \mathtt{C}_2 \xrightarrow{(cl,\iota)}_{\mathtt{ET}} (\mathcal{K},u,s),\mathtt{C}_i}$$

CIter
$$\frac{}{cl \vdash (\mathcal{K},u,s),\mathtt{C}^* \xrightarrow{(cl,\iota)}_{\mathtt{ET}} (\mathcal{K},u,s),\mathtt{skip} + (\mathtt{C}\,;\mathtt{C}^*)}$$

CSeqSkip
$$\frac{}{cl \vdash (\mathcal{K},u,s),\mathtt{skip}\,;\mathtt{C} \xrightarrow{(cl,\iota)}_{\mathtt{ET}} (\mathcal{K},u,s),\mathtt{C}}$$

CSeq
$$\frac{cl \vdash (\mathcal{K},u,s),\mathtt{C}_1 \xrightarrow{(cl,\iota)}_{\mathtt{ET}} (\mathcal{K},u',s'),\mathtt{C}_1'}{cl \vdash (\mathcal{K},u,s),\mathtt{C}_1\,;\mathtt{C}_2 \xrightarrow{(cl,\iota)}_{\mathtt{ET}} (\mathcal{K},u',s'),\mathtt{C}_1'\,;\mathtt{C}_2}$$

$\rightarrow$: (Conf $\times$ CEnv $\times$ Progs) $\times$ ET $\times$ Label $\times$ (Conf $\times$ CEnv $\times$ Progs)

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

Fig. 8: Operational Semantics on Key-value Store

ET	$\mathsf{can\text{-}commit}_{\mathtt{ET}}(\mathcal{K}, u, \mathcal{F})$	Closure Relation (where applicable)	$\mathsf{vshift}_{\mathtt{ET}}(\mathcal{K}, u, \mathcal{K}', u')$
MR	true		$u \sqsubseteq u'$
RYW	true		$\forall t \in \mathcal{K}' \setminus \mathcal{K}.\ \forall k, i.$ $\mathsf{w}(\mathcal{K}'(k,i)) \xrightarrow{\mathsf{SO}^?} t \Rightarrow i \in u'(k)$
MW	$\mathtt{closed}(\mathcal{K}, u, R_{\mathtt{MW}})$	$R_{\mathtt{MW}} \triangleq \mathsf{SO} \cap \mathsf{WW}_{\mathcal{K}}$	true
WFR	$\mathtt{closed}(\mathcal{K}, u, R_{\mathtt{WFR}})$	$R_{\mathtt{WFR}} \triangleq \mathsf{WR}_{\mathcal{K}}; (\mathsf{SO} \cup \mathsf{RW}_{\mathcal{K}})^?$	true
CC	$\mathtt{closed}(\mathcal{K}, u, R_{\mathtt{CC}})$	$R_{\mathtt{CC}} \triangleq \mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}$	$\mathsf{vshift}_{\mathtt{MR}\cap\mathtt{RYW}}(\mathcal{K}, u, \mathcal{K}', u')$
UA	$\mathtt{closed}(\mathcal{K}, u, R_{\mathtt{UA}})$	$R_{\mathtt{UA}} \triangleq \bigcup_{(u,k,\_)\in\mathcal{F}} \mathsf{WW}_{\mathcal{K}}^{-1}(k)$	true
PSI	$\mathtt{closed}(\mathcal{K}, u, R_{\mathtt{PSI}})$	$R_{\mathtt{PSI}} \triangleq R_{\mathtt{UA}} \cup R_{\mathtt{CC}} \cup \mathsf{WW}_{\mathcal{K}}$	$\mathsf{vshift}_{\mathtt{MR}\cap\mathtt{RYW}}(\mathcal{K}, u, \mathcal{K}', u')$
CP	$\mathtt{closed}(\mathcal{K}, u, R_{\mathtt{CP}})$	$R_{\mathtt{CP}} \triangleq \mathsf{SO}; \mathsf{RW}_{\mathcal{K}}^? \cup \mathsf{WR}_{\mathcal{K}}; \mathsf{RW}_{\mathcal{K}}^? \cup \mathsf{WW}_{\mathcal{K}}$	$\mathsf{vshift}_{\mathtt{MR}\cap\mathtt{RYW}}(\mathcal{K}, u, \mathcal{K}', u')$
WSI	$\mathtt{closed}(\mathcal{K}, u, R_{\mathtt{WSI}})$	$R_{\mathtt{SI}} \triangleq R_{\mathtt{UA}} \cup R_{\mathtt{CP}}$	$\mathsf{vshift}_{\mathtt{MR}\cap\mathtt{RYW}}(\mathcal{K}, u, \mathcal{K}', u')$
SI	$\mathtt{closed}(\mathcal{K}, u, R_{\mathtt{SI}})$	$R_{\mathtt{SI}} \triangleq R_{\mathtt{UA}} \cup R_{\mathtt{CP}} \cup (\mathsf{WW}_{\mathcal{K}}; \mathsf{RW}_{\mathcal{K}})$	$\mathsf{vshift}_{\mathtt{MR}\cap\mathtt{RYW}}(\mathcal{K}, u, \mathcal{K}', u')$
SER	$\mathtt{closed}(\mathcal{K}, u, R_{\mathtt{SER}})$	$R_{\mathtt{SER}} \triangleq \mathsf{WW}^{-1}$	true

Fig. 9: Execution tests of well-known consistency models where SO is as given in § 3.1.

B Relations to Dependency Graphs

Dependency graphs were introduced by Adya to define consistency models of transactional databases [1]. They are directed graphs consisting of transactions as nodes, each of which is labelled with transaction identifier and a set of read and write operations, and labelled edges between transactions for describing how information flows between nodes. Specifically, a transaction t reads a version for a key k that has been written by another transaction t' (*write-read dependency* WR), overwrites a version of k written by t' (*write-write dependency* WW), or reads a version of k that is later overwritten by t' (*read-write anti-dependency* RW). Note that we have named dependencies in kv-stores after the labelled edges of dependency graph. The main result of this Section shows that kv-stores are in fact bijective to dependency graphs, and dependencies in a kv-store naturally translates into a labelled edge in the associated dependency graph.

Definition 14. *A* dependency graph *is a quadruple* $\mathcal{G} = (\mathscr{T}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$*, where*

- $\mathscr{T} : \textsc{TransID} \rightharpoonup \mathcal{P}(\textsc{Ops})$ *is a partial mapping from transaction identifiers to the set of operations, where there are at most one read operation and one write operation per key, and such that* $\mathscr{T}(t_0) = \{(\mathtt{w}, k, v_0) \mid k \in \textsc{Keys}\}$*; furthermore,* $t_0 \in \mathrm{dom}(\mathscr{T}$*, and* $\mathscr{T}(t_0) = \{(\mathtt{w}, k, v_0) \mid k \in \textsc{Keys}\}$*,*
- $\mathsf{WR} : \textsc{Keys} \to \mathcal{P}(\mathrm{dom}(\mathscr{T}) \times \mathrm{dom}(\mathscr{T}))$ *is a function that maps each key* k *into a relation between transactions, such that for any* t, t_1, t_2, k, cl, m, n*:*
  - *if* $(\mathtt{r}, k, v) \in \mathscr{T}(t)$*, there exists* $t' \neq t$ *such that* $(\mathtt{w}, k, v) \in \mathscr{T}(t')$*, and* $t' \xrightarrow{\mathsf{WR}(k)} t$*,*
  - *if* $t_1 \xrightarrow{\mathsf{WR}(k)} t$ *and* $t_2 \xrightarrow{\mathsf{WR}(k)} t$*, then* $t_1 = t_2$*.*
  - *if* $t_{cl}^m \xrightarrow{\mathsf{WR}(k)} t_{cl}^n$*, then* $m < n$*.*
- $\mathsf{WW} : \textsc{Keys} \to \mathcal{P}(\mathrm{dom}(\mathscr{T}) \times \mathrm{dom}(\mathscr{T}))$ *is a function that maps each key into an irreflexive relation between transactions, such that for any* t, t', k, cl, m, n*,*
  - *if* $t \xrightarrow{\mathsf{WW}(k)} t'$*, then* $(\mathtt{w}, k, \_) \in \mathscr{T}(t), (\mathtt{w}, k, \_) \in \mathscr{T}(t')$*,*
  - *if* $(\mathtt{w}, k, \_) \in \mathscr{T}(t), (\mathtt{w}, k, \_) \in \mathscr{T}(t')$*, then either* $t = t'$*,* $t \xrightarrow{\mathsf{WW}(k)} t'$*, or* $t' \xrightarrow{\mathsf{WW}(k)} t$*; furthermore, if* $t = t_0$*, then it must be the case that* $t \xrightarrow{(\mathsf{WW}(k))} t'$*,*
  - *if* $t_{cl}^m \xrightarrow{\mathsf{WW}(k)} t_{cl}^n$*, then* $m < n$*,*
- $\mathsf{RW} : \textsc{Keys} \to \mathcal{P}(\mathrm{dom}(\mathscr{T}) \times \mathrm{dom}(\mathscr{T}))$ *is defined by letting* $t \xrightarrow{\mathsf{RW}(k)} t'$ *if and only if* $t'' \xrightarrow{\mathsf{WR}(k)} t$*,* $t'' \xrightarrow{\mathsf{WW}(k)} t'$ *for some* t''*.*

Let Dgraphs *be the set of all dependency graphs.*

Given a dependency graph $\mathcal{G} = (\mathscr{T}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$, we let $\mathsf{WR}_{\mathcal{G}} = \mathsf{WR}$, and similarly for WW and RW. We also let $T_{\mathcal{G}} = \mathrm{dom}(\mathscr{T})$, and write $(l, k, v) \in_{\mathcal{G}} t$ if $(;, k, v) \in \mathcal{G}(t)$. We often commit an abuse of notation and use WR to denote the relation $\bigcup_{k \in \textsc{Keys}} \mathsf{WR}(k)$; a similar notation is adopted for WW, RW. It will

always be clear from the context whether the symbol WR refers to a function from keys to relations, or to a relation between transactions.

As stated above, kv-stores are bijective to dependency graphs. The proof of this result is the topic of this Section.

Theorem 4. *There is a one-to-one map between kv-stores and dependency graphs.*

The proof structure of Theorem 4 is standard in its nature. We first how to encode a kv-store into a dependency graph. Then we show how to encode a dependency graph into a kv-store. Finally, we prove that the two constructions are one the inverse of the other: if we convert a kv-store $\mathcal{K}$ into a dependency graph $\mathcal{G}_{\mathcal{K}}$, then back to a kv-store $\mathcal{K}_{\mathcal{G}_{\mathcal{K}}}$, we obtain the initial kv-store.

To convert a kv-store $\mathcal{K}$ into a dependency graph, we first define how to extract a fingerprint of a transaction identifier t appearing in $\mathcal{K}$:

Definition 15. *Let $\mathcal{K}$ be a kv-store. For any transaction identifier t, we define $\mathcal{F}_{\mathcal{K}}(t)$ to be the smallest set such that whenever $\mathcal{K}(k, \_) = (v, t, \_)$ then $(\mathtt{w}, k, v) \in t$, and whenever $\mathcal{K}(k, \_) = (v, \_, \{t\} \cup \_)$, then $(\mathtt{r}, k, v) \in t$.*

Proposition 1. *For any $\mathcal{K}, t$, the fingerprint $\mathcal{F}_{\mathcal{K}}(t)$ is well defined. That is, whenever $(\mathtt{w}, k, v_1), (\mathtt{w}, k, v_2) \in \mathcal{F}_{\mathcal{K}}(t)$, then $v_1 = v_2$, and whenever $(\mathtt{r}, k, v_1), (\mathtt{r}, k, v_2) \in \mathcal{F}_{\mathcal{K}}(t)$, then $v_1 = v_2$.*

Proof. Suppose that $(\mathtt{w}, k, v_1), (\mathtt{w}, k, v_2) \in \mathcal{F}_{\mathcal{K}}(t)$ for some key, v_1, v_2. That is, there exist two indexes i_1, i_2 such that $\mathcal{K}(k, i_1) = (v_1, t, \_)$, and $\mathcal{K}(k, i_2) = (v_2, t, \_)$. That is, $\mathsf{w}(\mathcal{K}(k, i_1)) = \mathsf{w}(\mathcal{K}(k, i_2))$, and it follows from Def. 12 that $i_1 = i_2$. In particular, this implies that $v_1 = v_2$.

A similar argument can be used to prove that if $(\mathtt{r}, k, v_1), (\mathtt{r}, k, v_2) \in \mathcal{F}_{\mathcal{K}}(t)$, then $v_1 = v_2$. In this case, in fact, we have that there exist two indexes i_1, i_2 such that $\mathcal{K}(k, i_1) = (v_1, \_, \{t\} \cup \_)$, and $\mathcal{K}(k, i_2) = (v_2, \_, \{t\} \cup \_)$. Equivalently, $t \in \mathsf{rs}(\mathcal{K}(t, i_1)) \cap \mathsf{rs}(\mathcal{K}(t, i_2))$, and from Def. 12 it must be the case that $i_1 = i_2$, hence $v_1 = v_2$.

Using Def. 15, conerting a kv-store $\mathcal{K}$ into a dependency graph is immediate, as the following definition shows:

Definition 16. *Given a kv-store $\mathcal{K}$, the* dependency graph $\mathcal{G}_{\mathcal{K}} = (\mathscr{T}_{\mathcal{K}}, \mathsf{WR}_{\mathcal{K}}, \mathsf{WW}_{\mathcal{K}}, \mathsf{RW}_{\mathcal{K}})$ *is defined by letting $\mathscr{T}_{\mathcal{K}}(t)$ be defined if and only if $\mathcal{F}_{\mathcal{K}}(t) \neq \emptyset$, in which case we let $\mathscr{T}_{\mathcal{K}}(t) = \mathcal{F}_{\mathcal{K}}(t)$. The relations $\mathsf{WR}_{\mathcal{K}}, \mathsf{WW}_{\mathcal{K}}, \mathsf{RW}_{\mathcal{K}}$ are inherited directly from the transactional dependencies defined for $\mathcal{K}$.*

Definition 17. *Given a dependency graph $\mathcal{G} = (\mathscr{T}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$, we define the kv-store $\mathcal{K}_{\mathcal{G}}$ as follows:*

(1) for any transaction $t \in \mathrm{dom}(\mathscr{T})$ such that $(\mathtt{w}, k, v) \in \mathscr{T}(t)$, let $T = \left\{ t' \;\middle|\; t \xrightarrow{\mathsf{WR}(k)} t' \right\}$, and let $\nu(t, k) = (v, t, T)$,

(2) *For each key k, let $\nu_k^0 = (v_0, t_0, T_k^0)$, where* $T_k^0 = \left\{ t \;\middle|\; (\mathtt{r}, k, \_) \in \mathscr{T}(t) \wedge \forall t'.\ \neg(t' \xrightarrow{\mathsf{WR}(k)} t \right\}$. *Let also $\left\{\nu_k^i\right\}_{i=1}^n$ be the ordered set of versions such that, for any $i = 1, \cdots, n$, $\nu_k^i = \nu(t, k)$ for some t such that $(\mathtt{w}, k, \_) \in \mathscr{T}(t)$, and such that for any $i, j : 1 \leq i < j \leq n$, $\mathsf{w}(\nu_k^i) \xrightarrow{\mathsf{WW}(k)} \mathsf{w}(\nu_k^j)$. Then we let $\mathcal{K}_{\mathcal{G}} = \lambda k. \prod_{i=0}^{n} \nu_k^i$.*

Proposition 2. *Let $\mathcal{K}$ be a well-formed kv-store. Then $\mathcal{G}_{\mathcal{K}}$ is a well-formed dependency graph.*

Proof. Let $\mathcal{K}$ be a (well-formed) kv-store. We need to show that $\mathcal{G}_{\mathcal{K}} = (\mathscr{T}_{\mathcal{K}}, \mathsf{WR}_{\mathcal{K}}, \mathsf{WW}_{\mathcal{K}}, \mathsf{RW}_{\mathcal{K}})$ is a dependency graph. As a first step, we show that $\mathcal{G}_{\mathcal{K}}$ is a dependency graph, i.e. it satisfies all the constraints placed by Def. 14.

- Let $t \in \mathrm{dom}(\mathscr{T}_{\mathcal{K}})$, and suppose that $(\mathtt{r}, k, v) \in \mathscr{T}_{\mathcal{K}}(t)$. We need to prove that there exists a transaction $t' \in \mathrm{dom}(\mathscr{T}_{\mathcal{K}})$ such Because $(\mathtt{r}, k, v) \in \mathscr{T}_{\mathcal{K}}(t)$, there must exist an index $i : 0 \leq i < |\mathcal{K}(k)|$ such that $\mathcal{K}(k, i) = (v, t', \{t\} \cup \_)$ for some $t' \in \textsc{TransID}$. In this case we have that $t' \xrightarrow{\mathsf{WR}_{\mathcal{K}}(k)} t$, and by Def. 15 we have that $(\mathtt{w}, k, v) \in_{\mathcal{G}_{\mathcal{K}}} t'$.
- Let $t \in \mathrm{dom}(\mathscr{T}_{\mathcal{K}})$, and suppose that there exist t_1, t_2 such that $t_1 \xrightarrow{\mathsf{WR}_k(\mathcal{K})} t$, $t_2 \xrightarrow{\mathsf{WR}_k(\mathcal{K})} t$. By Def. 16, there exist two indexes $i, j : 0 \leq i, j < |\mathcal{K}(k)|$, such that $\mathcal{K}(k, i) = (\_, t_1, \{t\} \cup \_)$, $\mathcal{K}(k, j) = (\_, t_2, \{t\} \cup \_)$. We have that $t \in \mathsf{rs}(\mathcal{K}(k, i)) \cap \mathsf{rs}(\mathcal{K}(k, j))$, i.e. $\mathsf{rs}(\mathcal{K}(k, i)) \cap \mathsf{rs}(\mathcal{K}(k, j)) \neq \emptyset$. Because we are assuming that $\mathcal{K}$ is well-formed, then it must be the case that $i = j$. This implies that $t_1 = t_2$.
- Let $cl \in \textsc{ClientID}$, $m, n \in \mathbb{N}$ and $k \in \textsc{Keys}$ be such that $t_{cl}^n \xrightarrow{\mathsf{WR}_{\mathcal{K}}(k)} t_{cl}^m$. We prove that $n < m$. By Def. 16, it must be the case that there exists an index $i : 0 \leq i < |\mathcal{K}(k)|$ such that $\mathcal{K}(k, i) = (\_, t_{cl}^n, \{t_{cl}^m\} \cup \_)$. Because $\mathcal{K}$ is well-formed, it must be the case that $n < m$.
- Let $t \in \mathrm{dom}(\mathscr{T}_{\mathcal{K}})$. We show that $\neg(t \xrightarrow{\mathsf{WW}_{\mathcal{K}}} t)$. We prove this fact by contradiction: suppose that $t \xrightarrow{\mathsf{WW}_{\mathcal{K}}(k)} t$ for some key k. By Def. 16, there must exist two indexes $i, j : 0 \leq i < j < |\mathcal{K}(k)|$ such that $t = \mathsf{w}(\mathcal{K}(k, i))$ and $t = \mathsf{w}(\mathcal{K}(k, j))$. Because we are assuming that $\mathcal{K}$ is well-formed, then it must be the case that $i = j$, contradicting the statement that $i < j$.
- Let t, t' be such that $t' \xrightarrow{\mathsf{WW}_k(\mathcal{K})} t$. We must show that $(\mathtt{w}, k, \_) \in \mathscr{T}_{\mathcal{K}}(t')$, and $(\mathtt{w}, k, \_) \in \mathscr{T}_{\mathcal{K}}(t)$. By Def. 16, there exist $i, j : 0 \leq i, j < |\mathcal{K}(k)|$ such that $\mathcal{K}(k, i) = (v', t', \_)$ and $\mathcal{K}(k, j) = (v, t, \_)$, for some $v, v' \in \textsc{Val}$. Def. 16 also ensures that $(\mathtt{w}, k, v') \in \mathscr{T}_{\mathcal{K}}(t')$, and $(\mathtt{w}, k, v) \in \mathscr{T}_{\mathcal{K}}(t)$.
- Let t, t' be such that $(\mathtt{w}, k, \_) \in \mathscr{T}_{\mathcal{K}}(t)$ and $(\mathtt{w}, k, \_) \in \mathscr{T}_{\mathcal{K}}(t')$. We need to prove that either $t = t', t \xrightarrow{\mathsf{WW}_{\mathcal{K}}(k)} t'$, or $t' \xrightarrow{\mathsf{WW}_{\mathcal{K}}(k)} t$. By Def. 16 there exist two indexes $i, j : 0 < i, j < |\mathcal{K}(k)|$ such that $\mathcal{K}(k, i) = (\_, t, \_)$ and $\mathcal{K}(k, j) = (\_, t', \_)$. If $i = j$, then $t = t'$ and there is nothing left to prove. Otherwise, suppose without loss of generality that $i < j$. Then Def. 16 ensures that $t \xrightarrow{\mathsf{WW}_{\mathcal{K}}(k)} t'$.

– Suppose that $t^m_{cl} \xrightarrow{\mathsf{WW}_{\mathcal{K}}(k)} t^n_{cl}$ for some $cl \in \textsc{ClientID}$ and $m, n \in \mathbb{N}$. We need to show that $m < n$. By Def. 16, because $t^m_{cl} \xrightarrow{\mathsf{WW}_{\mathcal{K}}(k)} t^n_{cl}$ there exist two indexes $i, j : 0 < i, j < |\mathcal{K}(k)|$ such that $\mathsf{w}(\mathcal{K}(k,i)) = t^m_{cl}$ and $\mathsf{w}(\mathcal{K}(k,j)) = t^n_{cl}$. From the assumption that $\mathcal{K}$ is well-formed, it follows that $n < m$.

Next, we show how to convert a dependency graph $\mathcal{G}$ into a kv-store $\mathcal{K}$. The main idea is that any transaction $t \in T_{\mathcal{G}}$ induces a set of versions, and for each key k, the write-write-dependency order $\mathsf{WW}_{\mathcal{G}}(k)$ determines the order of these versions in $\mathcal{K}_{\mathcal{G}}$.

Definition 18. *Let $\mathcal{G}$ be a dependency graph. Given a key k, let n_k, $\{v^k_i\}^{n_k}_{i=0}$ $\{t^k_i\}^{n_k}_{i=0}$ be such that $\{t^k_i\}^{n_k}_{i=0} = \{t \mid (\mathtt{w}, k, v^k_i) \in_{\mathcal{G}} t\}$, where the index set $\{1, \cdots, n_k\}$ is chosen to be consistent with $\mathsf{WW}_{\mathcal{G}}(k)$: that is, $t_i \xrightarrow{\mathsf{WW}(k)} t_j$ if and only if $i < j$. Given a key k and an index $i = 1, \cdots, n_k$, we also let $T^k_i = \{t \mid t^k_i \xrightarrow{\mathsf{WR}(k)}\}t$. Note that this set is possibly empty. Finally, we let $\mathcal{K}_{\mathcal{G}}$ be such that, for any $k \in \textsc{Keys}$, $|\mathcal{K}_{\mathcal{G}}(k)| = n_k$, and for any $i = 0, \cdots, n$, $\mathcal{K}_{\mathcal{G}}(k,i) = (v^k_i, t^k_i, T^k_i)$.*

Proposition 3. *For any dependency graph $\mathcal{G}$, $\mathcal{K}_{\mathcal{G}}$ is a (well-formed) kv-store.*

Proof. We show that $\mathcal{K}_{\mathcal{G}}$ satisfies all the constraints fromf Def. 12. Throughout the proof, we adopt the same notation of Def. 18.

Let $k \in \textsc{Keys}$, and let i, j be such that $\mathsf{rs}(\mathcal{K}_{\mathcal{G}}(k,i)) \cap \mathsf{rs}(\mathcal{K}_{\mathcal{G}}(k,j)) \neq \emptyset$, that is there exists a transaction $t \in \mathsf{rs}(\mathcal{K}_{\mathcal{G}}(k,i)) \cap \mathsf{rs}(\mathcal{K}_{\mathcal{G}}(k,j))$. We show that $i = j$. By definition, $\mathsf{rs}(\mathcal{K}_{\mathcal{G}}(k,i)) = T^i_k$, and $\mathsf{rs}(\mathcal{K}_{\mathcal{G}}(k,j)) = T^j_k$. Def. 18 ensures that $t^k_i \xrightarrow{\mathsf{WR}_{\mathcal{G}}(k)} t$, and $t^k_j \xrightarrow{\mathsf{WR}_{\mathcal{G}}(k)} t$. By definition of dependency graph, it must be the case that $t^k_i = t^k_j$, and because the order of writers transactions in versions in $\mathcal{K}_{\mathcal{G}}(k)$ is defined to be consistent with $\mathsf{WW}_{\mathcal{G}}(k)$, then it must also be the case that $i = j$.

Suppose no that k, i, j are such that $\mathsf{w}(\mathcal{K}_{\mathcal{G}}(k,i)) = \mathsf{w}(\mathcal{K}_{\mathcal{G}}(k,j))$. By definition $\mathsf{w}(\mathcal{K}_{\mathcal{G}}(k,i)) = t^k_i$, and $\mathsf{w}(\mathcal{K}_{\mathcal{G}}(k,j) = t^k_j$. That is, $t^k_i = t^k_j$. Because the order of writer transactions in $\mathcal{K}_{\mathcal{G}}(k)$ is consistent with $\mathsf{WW}_{\mathcal{G}}(k)$, we also have that $i = j$.

Next, note that for any key k, $t^k_0 = t_0$. In fact, because $t_0 \in_{\mathcal{G}} (\mathtt{w}, k, v_0)$, we have that $t_0 = t^k_i$ for some $i = 0, \cdots, n_k$. Also, because whenever t is such that $(\mathtt{w}, k, \_) \in_{\mathcal{G}} t$, then it must be the case that $t_0 \xrightarrow{\mathsf{WW}(k)} t$, then it must be the case that $i = 0$. It follows that, for any $k \in \textsc{Keys}$, $\mathcal{K}_{\mathcal{G}}(k,0) = (v_0, t_0, \_)$.

Finally, suppose that $t^n_{cl} = \mathsf{w}(\mathcal{K}_{\mathcal{G}}(k,i))$, $t^m_{cl} = \mathsf{w}(\mathcal{K}_{\mathcal{G}}(k,j))$ for some i, j such that $i < j$. In this case we have that $t^n_{cl} = t^k_i$, $t^m_{cl} = t^k_j$, and because $i < j$ it must be the case that $t^n_{cl} \xrightarrow{\mathsf{WW}_{\mathcal{G}}(k)} t^m_{cl}$. The definition of dependency graph ensures then that it must $n < m$. A similar argument shows that, if $t^n_{cl} \in \mathsf{w}(\mathcal{K}_{\mathcal{G}}(k,i)), t^m_{cl} \in \mathsf{rs}(\mathcal{K}_{\mathcal{G}}(k,i))$, then it must be the case that $t^n_{cl} \xrightarrow{\mathsf{WR}_{\mathcal{G}}(k)} t^m_{cl}$, and therefore $n < m$.

Finally, we need to show that the two constructions outlined in Def. 18 and Def. 17 are one the inverse of the other.

Proposition 4. *For any kv-store $\mathcal{K}$, $\mathcal{K}_{\mathcal{G}_{\mathcal{K}}} = \mathcal{K}$.*

Proof. We prove that for any $k \in \textsc{Keys}$, $\mathcal{K}(k) = \mathcal{K}_{\mathcal{G}_\mathcal{K}}(k)$.

Let then $k \in \textsc{Keys}$, and suppose that $\mathcal{K}(k) = (v_0, t_0, T_0) \cdots (v_n, t_n, T_n)$. By construction, in $(\mathtt{w}, k, v_i) \in_{\mathcal{G}_\mathcal{K}} t_i$, and whenever there is a transaction t such that $(\mathtt{w}, k, t) \in_{\mathcal{G}_\mathcal{K}} t$, then $t = t_i$ for some $i = 0, \cdots, n$. In particular, we have that $t_0 \xrightarrow{\mathsf{WW}_\mathcal{K}(k)} \cdots \xrightarrow{\mathsf{WW}_\mathcal{K}(k)} t_n$ completely characterises the write-write-dependency relation $\mathsf{WW}_\mathcal{K}(k)$ over $\mathcal{K}_\mathcal{G}$ (recall that, by Def. 18, $\mathsf{WR}_{\mathcal{G}_\mathcal{K}} = \mathsf{WR}_\mathcal{K}$). By definition, we have that $\mathcal{K}_{\mathcal{G}_\mathcal{K}} = (v_0, t_0, T'_0) \cdots (v_n, t_n, T'_n)$.

It remains to prove that, for any $i = 0, \cdots, n$, $T'_i = T_i$. For any $i = 0, \cdots, n$, and transaction $t \in T_i$, Def. 18 ensures that $t_i \xrightarrow{\mathsf{WR}_\mathcal{K}} t$, and by Def. 17 it must be the case that $t \in T'_i$. Furthermore, if $t' \in T'_i$, then from Def. 17 it must be the case that $t_i \xrightarrow{\mathsf{WR}_{\mathcal{G}_{mkvs}}(k)} t'$, or equivalently $t_i \xrightarrow{\mathsf{WR}_\mathcal{K}(k)} t'_i$. (Def. 18). Then it must be the case that $t' \in T_i$.

Proposition 5. *For any dependency graph $\mathcal{G}$, $\mathcal{G}_{\mathcal{K}_\mathcal{G}} = \mathcal{G}$.*

Proof. The proof of this claim is similar to Prop. 4, and therefore omitted.

C Operational Semantics of Abstract Executions

Abstract executions are a framework originally introduced in [13] to capture the run-time behaviour of clients interacting with a database. In abstract execution, two relations between transactions are introduced: the *visibility* relation establishes when a transaction observes the effects of another transaction; and the *arbitration* relation helps to determine the value of a key k read by a transaction, in the case that the transaction observes multiple updates to k performed by different transactions.

Definition 19. *An abstract execution is a triple* $\mathcal{X} = (\mathscr{T}, \mathsf{VIS}, \mathsf{AR})$*, where*

- $\mathscr{T} : \textsc{TransID} \rightharpoonup \mathcal{P}(\textsc{Ops})$ *is a partial, finite function mapping transaction identifiers to the set of operations that they perform, with* $\mathscr{T}(t_0) = \{(\mathtt{w}, k, v_0 \mid k \in \textsc{Keys}\}$*,*
- $\mathsf{VIS} \subseteq \mathrm{dom}(\mathscr{T}) \times \mathrm{dom}(\mathscr{T})$ *is an irreflexive relation, called* visibility*,*
- $\mathsf{AR} \subseteq \mathrm{dom}(\mathscr{T}) \times \mathrm{dom}(\mathscr{T})$ *is a strict, total order such that* $\mathsf{VIS} \subseteq \mathsf{AR}$*, and whenever* $t^n_{cl} \xrightarrow{\mathsf{AR}} t^m_{cl}$*, then* $n < m$*.*

The set of abstract executions is denoted by absExec*.*

Given an abstract execution $\mathcal{X} = (\mathscr{T}, \mathsf{VIS}, \mathsf{AR})$, the notation $\mathscr{T}_{\mathcal{X}} = \mathscr{T}$, $T_{\mathcal{X}} = \mathrm{dom}(\mathscr{T})$, $\mathsf{VIS}_{\mathcal{X}} = \mathsf{VIS}$ and $\mathsf{AR}_{\mathcal{X}} = \mathsf{AR}$. The session order for a client $\mathsf{SO}_{\mathcal{X}}(cl)$ and then the overall session order $\mathsf{SO}_{\mathcal{X}}$ are defined as the following:

$$\mathsf{SO}_{\mathcal{X}}(cl) = \{(t^n_{cl}, t^m_{cl}) \mid cl \in \textsc{ClientID} \wedge t^n_{cl} \in T_{\mathcal{X}} \wedge t^m_{cl} \in T_{\mathcal{X}} \wedge n < m\}$$

and

$$\mathsf{SO}_{\mathcal{X}} = \bigcup_{cl \in \textsc{ClientID}} \mathsf{SO}_{\mathcal{X}}(cl)$$

The notation $(\mathtt{r}, k, v) \in_{\mathcal{X}} t$ denotes $(\mathtt{r}, k, v) \in \mathscr{T}_{\mathcal{X}}(t)$, and similarly for write operations $(\mathtt{w}, k, v) \in_{\mathcal{X}} t$. Given an abstract execution $\mathcal{X}$, a transaction $t \in T_{\mathcal{X}}$, and a key k, the visible writers set $\mathsf{visibleWrites}_{\mathcal{X}}(k, t) \triangleq \left\{ t' \;\middle|\; t' \xrightarrow{\mathsf{VIS}_{\mathcal{X}}} t \wedge (\mathtt{w}, k, \_) \in_{\mathcal{X}} t' \right\}$.

The operational semantics on abstract executions (Fig. 10) is parametrised in the axiomatic definition $(\mathsf{RP}, \mathcal{A})$ of a consistency model: transitions take the form $(\mathcal{X}, \textsc{CEnv}, \mathtt{P}) \xrightarrow{}_{(\mathsf{RP},\mathcal{A})} (\mathcal{X}', \textsc{CEnv}', \mathtt{P}')$. An axiomatic definition of a consistency model is given by a pair $(\mathsf{RP}, \mathcal{A})$, where RP is a resolution policy (Def. 20) and $\mathcal{A}$ is a set of axioms for visibility relation (Def. 21). An abstract execution $\mathcal{X}$ satisfies the consistency model, written $\mathcal{X} \models (\mathsf{RP}, \mathcal{A})$ if it satisfies its individual components. The set of abstract executions induced by an axiomatic definition is given by $\mathsf{CM}(\mathsf{RP}, \mathcal{A}) = \{\mathcal{X} \mid \mathcal{X} \models (\mathsf{RP}, \mathcal{A})\}$.

We first introduce a notation of two abstract executions *agree*. Given two abstract executions $\mathcal{X}_1, \mathcal{X}_2 \in \mathsf{absExec}$ and set of transactions $T \subseteq T_{\mathcal{X}_1} \cap T_{\mathcal{X}_2}$, $\mathcal{X}_1$ and $\mathcal{X}_2$ *agree* on T if and only if for any transactions t t' in T:

$$\mathscr{T}_{\mathcal{X}_1}(t) = \mathscr{T}_{\mathcal{X}_2}(t) \wedge ((t \xrightarrow{\mathsf{VIS}_{\mathcal{X}_1}} t') \Leftrightarrow (t \xrightarrow{\mathsf{VIS}_{\mathcal{X}_2}} t')) \wedge ((t \xrightarrow{\mathsf{AR}_{\mathcal{X}_1}} t') \Leftrightarrow (t \xrightarrow{\mathsf{AR}_{\mathcal{X}_2}} t'))$$

Definition 20. *A resolution policy* RP *is a function* $\mathsf{RP} : \mathsf{absExec} \times \mathcal{P}(\textsc{TransID}) \to \mathcal{P}(\textsc{Snapshot})$ *such that, for any* $\mathcal{X}_1, \mathcal{X}_2$ *that agree on a subset of transactions* T*, then* $\mathsf{RP}(\mathcal{X}_1, T) = \mathsf{RP}(\mathcal{X}_2, T)$*. An abstract execution* $\mathcal{X}$ *satisfies the execution policy* RP *if,*

$$\forall t \in T_{\mathcal{X}}.\ \exists \sigma \in \mathsf{RP}(\mathcal{X}, \mathsf{VIS}_{\mathcal{X}}^{-1}(t)).\ \forall k, v.\ (\mathtt{r}, k, v) \in_{\mathcal{X}} t \Rightarrow \sigma(k) = v$$

Definition 21. *An axiom* A *is a function from abstract executions to relations between transactions,* $\mathsf{A} : \mathsf{absExec} \to \mathcal{P}(\textsc{TransID} \times \textsc{TransID})$*, such that whenever* $\mathcal{X}_1, \mathcal{X}_2$ *agree on a subset of transactions* T*, then* $\mathsf{A}(\mathcal{X}_1) \cap (T \times T) \subseteq \mathsf{A}(\mathcal{X}_2)$*.*

Axioms of a consistency model are constraints of the form $\mathsf{A}(\mathcal{X}) \subseteq \mathsf{VIS}_{\mathcal{X}}$. For example, if we require $A(\mathcal{X}) = \mathsf{AR}_{\mathcal{X}}$, then the corresponding axiom is given by $\mathsf{AR}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$, thus capturing the serialisability of transactions, i.e. this axiom is equivalent to require that $\mathsf{VIS}_{\mathcal{X}}$ is a total order. The requirement on subsets of transactions on which abstract executions agree will be needed later, when we define an operational semantics of transactions where clients can append a new transaction t at the tail of an abstract execution $\mathcal{X}$, which satisfies an axiom A. This requirement ensures that we only need to check that the axiom is A is satisfied by the pre-visibility and pre-arbitration relation of the transaction t in $\mathcal{X}'$. In fact, the resulting abstract execution $\mathcal{X}'$ agrees with $\mathcal{X}$ on the set $T_{\mathcal{X}}$: in this case we'll note that we can rewrite $\mathsf{A}(\mathcal{X}') = \mathsf{A}(\mathcal{X}') \cap ((T_{\mathcal{X}} \times T_{\mathcal{X}}) \cup (T_{\mathcal{X}} \times \{t\}))$. Then $\mathsf{A}(\mathcal{X}') \cap ((T_{\mathcal{X}} \times T_{\mathcal{X}})) \subseteq \mathsf{A}(\mathcal{X}) \cap (T_{\mathcal{X}} \times T_{\mathcal{X}}) \subseteq \mathsf{VIS}_{\mathcal{X}} \cap (T_{\mathcal{X}} \times T_{\mathcal{X}}) \subseteq \mathsf{VIS}_{\mathcal{X}'}$, hence we only need to check that $\mathsf{A}(\mathcal{X}') \cap (T_{\mathcal{X}} \times \{t\}) \subseteq \mathsf{VIS}_{\mathcal{X}'}$.

We say that an abstract execution $\mathcal{X}$ satisfies an axiom A, if $\mathsf{A}(\mathcal{X}) \subseteq \mathsf{VIS}_{\mathcal{X}}$. An abstract execution $\mathcal{X}$ satisfies $(\mathsf{RP}, \mathcal{A})$, written $\mathcal{X} \models (\mathsf{RP}, \mathcal{A})$, if the abstract execution $\mathcal{X}$ satisfies RP and $\mathcal{A}$.

Definition 22 (Abstract executions induced by axiomatic definition). *The set of all abstract executions induced by an axiomatic definition,* $\mathsf{CM}(\mathsf{RP}, \mathcal{A})$ *is defined as* $\mathsf{CM}(\mathsf{RP}, \mathcal{A}) \triangleq \{\mathcal{X} \mid \mathcal{X} \models (\mathsf{RP}, \mathcal{A})\}$*.*

The Fig. 10 presents all rules of the operational semantics of programs based on abstract executions. The ACommit rule is the abstract execution counterpart of rule PCommit for kv-stores. The ACommit models how an abstract execution $\mathcal{X}$ evolves when a client wants to execute a transaction whose code is $\llbracket \mathtt{T} \rrbracket$. In the rule, T is the set of transactions of $\mathcal{X}$ that are visible to the client cl that wishes to execute $\llbracket \mathtt{T} \rrbracket$. Such a set of transactions is used to determine a snapshot $\sigma \in \mathsf{RP}(\mathcal{X}, T)$ that the client cl uses to execute the code $\llbracket \mathtt{T} \rrbracket$, and obtain a fingerprint $\mathcal{F}$. This fingerprint is then used to extend abstract execution $\mathcal{X}$ with a transaction from the set $\mathsf{nextTid}(T_{\mathcal{X}}, cl)$. Similar PProg rule, the AProg rule in Fig. 10 models multi-clients concurrency in an interleaving fashion. All the rest rules of the abstract operational semantics in Fig. 10 have a similar counterpart in the kv-store semantics.

Note that AAtomicTrans is more general than Rule PAtomicTrans in the kv-store semantics. In the latter, the snapshot of a transaction is uniquely determined from a view of the client, in a way that roughly corresponds to the

$\rightarrow$: ClientID $\times$ ((AbsExecs $\times$ Stack) $\times$ Commands) $\times$ ET $\times$ Label $\times$ ((AbsExecs $\times$ Stack) $\times$ Commands)

AAtomicTrans
$$\frac{T \subseteq T_{\mathcal{X}} \qquad \sigma \in \mathsf{RP}(\mathcal{X}, T) \qquad (s, \sigma, \emptyset), \mathtt{T} \rightarrow^{*} (s', \_, \mathcal{F}), \mathtt{skip} \qquad t \in \mathsf{nextTid}(T_{\mathcal{X}}, cl) \qquad \mathcal{X}' = \mathsf{extend}(\mathcal{X}, t, T, \mathcal{F}) \qquad \forall A \in \mathcal{A}.\ \{t' \mid (t', t) \in \mathsf{A}(\mathcal{X}')\} \subseteq T}{cl \vdash (\mathcal{X}, s), [\mathtt{T}] \xrightarrow{(cl, T, \mathcal{F})}_{(\mathsf{RP}, \mathcal{A})} (\mathcal{X}', s'), \mathtt{skip}}$$

APrimitive
$$\frac{s \overset{\mathtt{C}_p}{\rightsquigarrow} s'}{cl \vdash (\mathcal{X}, s), \mathtt{C}_p \xrightarrow{(cl, \iota)}_{\mathtt{ET}} (\mathcal{X}, s'), \mathtt{skip}}$$

AChoice
$$\frac{i \in \{1, 2\}}{cl \vdash (\mathcal{X}, s), \mathtt{C}_1 + \mathtt{C}_2 \xrightarrow{(cl, \iota)}_{\mathtt{ET}} (\mathcal{X}, s), \mathtt{C}_i}$$

AIter
$$\frac{}{cl \vdash (\mathcal{X}, s), \mathtt{C}^{*} \xrightarrow{(cl, \iota)}_{\mathtt{ET}} (\mathcal{X}, s), \mathtt{skip} + (\mathtt{C}\,;\mathtt{C}^{*})}$$

ASeqSkip
$$\frac{}{cl \vdash (\mathcal{X}, s), \mathtt{skip}\,;\mathtt{C} \xrightarrow{(cl, \iota)}_{\mathtt{ET}} (\mathcal{X}, s), \mathtt{C}}$$

ASeq
$$\frac{cl \vdash (\mathcal{X}, s), \mathtt{C}_1 \xrightarrow{(cl, \iota)}_{\mathtt{ET}} (\mathcal{X}, s'), \mathtt{C}_1{}'}{cl \vdash (\mathcal{X}, s), \mathtt{C}_1\,;\mathtt{C}_2 \xrightarrow{(cl, \iota)}_{\mathtt{ET}} (\mathcal{X}, s'), \mathtt{C}_1{}'\,;\mathtt{C}_2}$$

$\rightarrow$: (AbsExecs $\times$ CEnv $\times$ Progs) $\times$ ET $\times$ Label $\times$ (AbsExecs $\times$ CEnv $\times$ Progs)

AProg
$$\frac{cl \vdash (\mathcal{X}, \mathcal{E}(cl)), \mathtt{P}\,(cl)\,, \xrightarrow{\lambda}_{(\mathsf{RP}, \mathcal{A})} (\mathcal{X}', s'), \mathtt{C}'}{(\mathcal{X}, \mathcal{E}), \mathtt{P} \xrightarrow{\lambda}_{(\mathsf{RP}, \mathcal{A})} (\mathcal{X}', \mathcal{E}[cl \mapsto s']), \mathtt{P}[cl \mapsto \mathtt{C}']}$$

Fig. 10: Operational Semantics on Abstract Executions

last write wins policy in the abstract execution framework. In contrast, the snapshot of a transaction used in AAtomicTrans is chosen non-deterministically from those made available to the client by the resolution policy RP, which may not necessarily be last-write-win.

Throughout this report we will work mainly with the *Last Write Wins* resolution policy (Def. 23). When discussing the operational semantics of transactional programs, we will also introduce the *Anarchic* resolution policy.

Definition 23. *The Last Write Wins resolution policy* $\mathsf{RP}_{\mathsf{LWW}}$ *is defined as* $\mathsf{RP}_{\mathsf{LWW}}(\mathcal{X}, T) \triangleq \{\sigma\}$ *where*

$$\sigma = \lambda k.let\ T_k = (T \cap \{t \mid (\mathtt{w}, k, \_) \in_{\mathcal{X}} t\})\ in\ \begin{cases} v_0 & if\ T_k = \emptyset \\ v & if\ (\mathtt{w}, k, v) \in_{\mathcal{X}} \max_{\mathsf{AR}_{\mathcal{X}}}(T_k) \end{cases}$$

D Relationship between kv-stores and abstract execution

D.1 KV-Store to Abstract Executions

We introduce the definition of the dependency graph induced an abstract execution:

Definition 24. *Given an abstract execution $\mathcal{X}$ that satisfies the last write wins policy, the dependency graph* $\mathsf{graphOf}(\mathcal{X}) \triangleq (\mathscr{T}_{\mathcal{X}}, \mathsf{WR}_{\mathcal{X}}, \mathsf{WW}_{\mathcal{X}}, \mathsf{RW}_{\mathcal{X}})$ *is defined by letting*

- $t \xrightarrow{\mathsf{WR}_{\mathcal{X}}(k)} t'$ *if and only if* $t = \max_{\mathsf{AR}_{\mathcal{X}}}(\mathsf{visibleWrites}_{\mathcal{X}}(k, t'))$,
- $t \xrightarrow{\mathsf{WW}_{\mathcal{X}}(k)} t'$ *if and only* $t, t' \in_{\mathcal{X}} (\mathtt{w},\ k, \_)$ *and* $t \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t'$,
- $t \xrightarrow{\mathsf{RW}_{\mathcal{X}}(k)} t'$ *if and only if either* $(\mathtt{r}, k, \_) \in_{\mathcal{X}} t, (\mathtt{w}, k, \_) \in_{\mathcal{X}} t'$ *and whenever* $t'' \xrightarrow{\mathsf{WR}_{\mathcal{X}}(k)} t$, *then* $t'' \xrightarrow{\mathsf{WW}_{\mathcal{X}}(k)} t'$.

Note that each abstract execution $\mathcal{X}$ determines a kv-store $\mathcal{K}_{\mathcal{X}}$, as a result of Def. 24 and Theorem 4. Let $\mathcal{K}$ be the unique kv-store such that $\mathcal{G}_{\mathcal{K}} = \mathsf{graphOf}(\mathcal{X})$, then $\mathcal{K}_{\mathcal{X}} = \mathcal{K}$. As we will discuss later in this Section, this mapping $\mathcal{K}_{(\_)}$ is NOT a bijection, in that several abstract executions may be encoded in the same kv-store. Because kv-stores abstract away the total arbitration order of transactions.

Upon the relation $\mathcal{K}_{\mathcal{X}} = \mathcal{K}$, there is a deeper link between kv-store plus views and abstract exertions. This notion, named *compatibility*, bases on the intuition that clients can make observations over kv-stores and abstract executions, in terms of snapshots.

In kv-stores, observations are snapshots induced by views. While in abstract executions, observations correspond to the snapshots induced by the visible transactions. Note that it is under the condition that the abstract execution adopts $\mathsf{RP}_{\mathsf{LWW}}$ resolution policy. This approach is analogous to the one used by operation contexts in [14]. Thus, a kv-store $\mathcal{K}$ is *compatible* with an abstract execution $\mathcal{X}$, written $\mathcal{K} \simeq \mathcal{X}$ if any observation made on $\mathcal{K}$ can be replicated by an observation made on $\mathcal{X}$, and vice-versa.

Definition 25. *Given a kv-store $\mathcal{K}$, an abstract execution $\mathcal{X}$ is compatible with $\mathcal{K}$, written $\mathcal{X} \simeq \mathcal{K}$, if and only if there exists a mapping $f : \mathcal{P}(T_{\mathcal{X}}) \to \textsc{Views}(\mathcal{K})$ such that*

- *for any subset $T \subseteq T_{\mathcal{X}}$, then $\mathsf{RP}_{\mathsf{LWW}}(\mathcal{X}, T) = \{\mathsf{snapshot}(\mathcal{K}, f(T))\}$;*
- *for any view $u \in \textsc{Views}(\mathcal{K})$, there exists a subset $T \subseteq T_{\mathcal{X}}$ such that $f(T) = u$, and $\mathsf{RP}_{\mathsf{LWW}}(\mathcal{X}, T) = \{\mathsf{snapshot}(\mathcal{K}_{\mathcal{X}}, u)\}$.*

The function $\mathsf{getView}(\mathcal{X}, T)$ defines the view on $\mathcal{K}_{\mathcal{X}}$ that corresponds to T as the following:

$$\mathsf{getView}(\mathcal{X}, T) \triangleq \lambda k.\, \{0\} \cup \{i \mid \mathsf{w}(\mathcal{K}_{\mathcal{X}}(k, i)) \in T\}$$

Inversely, the function $\mathsf{visTx}(\mathcal{K}, u)$ converts a view to a set of observable transactions:

$$\mathsf{visTx}(\mathcal{K}, u) \triangleq \{\mathsf{w}(\mathcal{K}(k, i)) \mid k \in \textsc{Keys} \wedge i \in u\,(k)\}$$

Given $\mathsf{getView}$, visTx, Def. 25, it follows $\mathcal{X} \simeq \mathcal{K}_{\mathcal{X}}$ shown in Theorem 5.

Theorem 5. *For any abstract execution $\mathcal{X}$ that satisfies the last write wins policy, $\mathcal{X} \simeq \mathcal{K}_{\mathcal{X}}$.*

Proof. Given the function $\mathsf{getView}(\mathcal{X}, \cdot)$ from $\mathcal{P}\,(T_{\mathcal{X}})$ to $\textsc{Views}\,(\mathcal{K}_{\mathcal{X}})$, we prove it satisfies the constraint of Def. 25. Fix a set of transitions T. By the Prop. 6, the view $\mathsf{getView}(\mathcal{X}, T)$ on $\mathcal{K}_{\mathcal{X}}$ is a valid view, that is, $\mathsf{getView}(\mathcal{X}, T) \in \textsc{Views}\,(\mathcal{K}_{\mathcal{X}})$. Given that it is a valid view, the Prop. 7 proves:

$$\mathsf{RP}_{\mathsf{LWW}}(\mathcal{X}, T) = \{\mathsf{snapshot}(\mathcal{K}_{\mathcal{X}}, \mathsf{getView}(\mathcal{X}, T))\} \tag{4.1}$$

The another way round is more subtle, because T contains any read only transaction. By Prop. 8, it is safe to erase read only transactions from T, when calculating the view $\mathsf{getView}(\mathcal{X}, T)$. Last, by Prop. 9, we prove the following:

$$\mathsf{RP}_{\mathsf{LWW}}(\mathcal{X}, T) = \mathsf{snapshot}(\mathcal{K}_{\mathcal{X}}, u) \tag{4.2}$$

By Eq. (4.1) and Eq. (4.2), it follows $\mathcal{X} \simeq \mathcal{K}_{\mathcal{X}}$.

Proposition 6 (Valid views). *For any abstract execution $\mathcal{X}$, and $T \subseteq T_{\mathcal{X}}$, $\mathsf{getView}(\mathcal{X}, T) \in \textsc{Views}\,(\mathcal{K}_{\mathcal{X}})$.*

Proof. Assume an abstract execution $\mathcal{X}$, a set of transactions $T \subseteq T_{\mathcal{X}}$, and a key k. By the definition of $\mathsf{getView}(\mathcal{X}, T)$, then $0 \in \mathsf{getView}(\mathcal{X}, T)(k)$, and $0 \leq i < |\mathcal{K}_{\mathcal{X}}(k)|$ for any index i such that $i \in \mathsf{getView}(\mathcal{X}, T)(k)$. Therefore we only need to prove that $\mathsf{getView}(\mathcal{X}, T)$ satisfies (atomic). Let $j \in \mathsf{getView}(\mathcal{X}, T)(k)$ for some key k, and let $t = \mathsf{w}(\mathcal{K}_{\mathcal{X}}(k, j))$. Let also k', i be such that $\mathsf{w}(\mathcal{K}_{\mathcal{X}}(k', i)) = t$. We need to show that $i \in \mathsf{getView}(\mathcal{X}, T)(k')$. Note that it $t = t_0$ then $\mathsf{w}(\mathcal{K}_{\mathcal{X}}(k', i)) = t$ only if $i = 0$, and $0 \in \mathsf{getView}(\mathcal{X}, T)(k')$ by definition. Let then $t \neq t_0$. Because $\mathsf{w}(\mathcal{K}_{\mathcal{X}}(k, j)) = t$ and $j \in \mathsf{getView}(\mathcal{X}, T)$, then it must be the case that $t \in T$. Also, because $\mathsf{w}(\mathcal{K}_{\mathcal{X}}(k', i)) = t$, then $(\mathtt{w}, k, \_) \in \mathscr{T}_{\mathcal{X}}(t)$. It follows that there exists an index $i' \in \mathsf{getView}(\mathcal{X}, t)(k')$ such that $\mathsf{w}(\mathcal{K}_{\mathcal{X}}(k', i')) = t$. By definition of $\mathcal{K}_{\mathcal{X}}$, if $\mathsf{w}(\mathcal{K}_{\mathcal{X}}(k', i')) = t$, then it must be $i' = i$, and therefore $i \in \mathsf{getView}(\mathcal{X}, t)(k')$.

Proposition 7 (Visible transactions to views). *For any subset $T \subseteq T_{\mathcal{X}}$, $\mathsf{RP}_{\mathsf{LWW}}(\mathcal{X}, T) = \{\mathsf{snapshot}(\mathcal{K}_{\mathcal{X}}, \mathsf{getView}(\mathcal{X}, T))\}$.*

Proof. Fix $T \subseteq \mathcal{X}$, and let $\{\mathcal{K}\} = \mathsf{RP}_{\mathsf{LWW}}(\mathcal{X}, T)$. We prove that, for any $k \in \textsc{Keys}$, $\mathcal{K}(k) = \mathsf{snapshot}(\mathsf{getView}(\mathcal{X}, T))(k)$. There are two different cases:

(1) $T \cap \{t \mid (\mathtt{w}, k, \_) \in_{\mathcal{X}} t\} = \emptyset$. In this case $\mathcal{K}(k) = v_0$. We know that $\mathsf{graphOf}(\mathcal{X})$ satisfies all the constraints required by the definition of dependency graph ([18]). Together with Theorem 4 it follows that $\mathcal{K}_{\mathcal{X}}(k, 0) = (v_0, t_0, \_)$. We prove that $\mathsf{getView}(\mathcal{X}, T)(k) = \{0\}$, hence

$$\mathsf{snapshot}(\mathcal{K}_{\mathcal{X}}, \mathsf{getView}(\mathcal{X}, T))(k) = \mathsf{val}(\mathcal{K}_{\mathcal{X}}(k, 0)) = v_0$$

Note that whenever $(\mathtt{w}, k, \_) \in_{\mathcal{X}} t$ for some t, then $t \notin T$. Therefore, whenever $(v, t, \_) = \mathcal{K}_{\mathcal{X}}(k, i)$ for some $i \geq 0$, then $t \notin T$.

$$\mathsf{getView}(\mathcal{X}, T)(k) = \{0\} \cup \{i \mid \mathsf{w}(\mathcal{K}_{\mathcal{X}}(k, i)) \in T)\} = \{0\} \cup \emptyset = \{0\}$$

(2) Suppose now that $T \cap \{t \mid (\mathtt{w}, k, \_) \in_{\mathcal{X}} t\} \neq \emptyset$. Let then $t = \max_{\mathsf{AR}_{\mathcal{X}}}(T \cap \{t \mid (\mathtt{w}, k, \_) \in_{\mathcal{X}} t\})$. Then $(\mathtt{w}, k, v) \in_{\mathcal{X}} t$ for some $v \in \textsc{Val}$. Furthermore, $\mathsf{RP}_{\mathsf{LWW}}(\mathcal{X}, T)(k) = v$. By definition, $t' \in T \cap \{t \mid (\mathtt{w}, k, \_) \in_{\mathcal{X}} t\}$, then either $t' = t$ or $t' \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t$. The definition of $\mathsf{graphOf}(\mathcal{X})$ gives that $t' \xrightarrow{\mathsf{WW}_{\mathcal{X}}(k)} t$. Because $(\mathtt{w}, k, v) \in_{\mathcal{X}} t$, then there exists an index $i \geq 0$ such that $\mathcal{K}_{\mathcal{X}}(k, i) = (v, t, \_)$. Furthermore, whenever $\mathsf{w}(k, j) = t'$ for some t' and $j > i$, then it must be the case that $t \xrightarrow{\mathsf{WW}_{\mathcal{X}}(k)} t'$, and because $\mathsf{WW}_{\mathcal{X}}(k)$ is transitive and irreflexive, it must be that $\neg(t' \xrightarrow{\mathsf{WW}_{\mathcal{X}}(k)} t)$ and $t \neq t'$: this implies that $t' \notin T$. It follows that $\max(\mathsf{getView}(\mathcal{X}, T)(k)) = i$, hence $\mathsf{snapshot}(\mathcal{K}_{\mathcal{X}}, \mathsf{getView}(\mathcal{X}, T)) = \mathsf{val}(\mathcal{K}_{\mathcal{X}}(k, i)) = v$.

Proposition 8 (Read-only transactions erasing). *Let $u \in \textsc{Views}(\mathcal{K}_{\mathcal{X}})$, and let $T \subseteq T_{\mathcal{X}}$ be a set of read-only transactions in $\mathcal{X}$. Then* $\mathsf{getView}(\mathcal{X}, T \cup \mathsf{visTx}(\mathcal{K}_{\mathcal{X}}, u)) = u$.

Proof. Fix a key k. Suppose that $i \in \mathsf{getView}(\mathcal{X}, T \cup \mathsf{visTx}(\mathcal{K}_{\mathcal{X}}, u))(k)$. By definition, $\mathcal{K}_{\mathcal{X}}(k, j) = (\_, t, \_)$ for some $t \in T \cup \mathsf{visTx}(\mathcal{K}_{\mathcal{X}}, u)$. Because T only contains read-only transactions, by definition of $\mathcal{K}_{\mathcal{X}}$ there exists no index j such that $\mathcal{K}_{\mathcal{X}}(k, j) = (\_, t', \_)$ for some $t' \in T$, hence it must be the case that $t \in \mathsf{visTx}(\mathcal{K}_{\mathcal{X}}, u)$. By definition of visTx, this is possible only if there exist a key k' and an index j such that $\mathcal{K}_{\mathcal{X}}(k', u) = (\_, t, \_)$. Because u is atomic by definition, and because $\mathcal{K}_{\mathcal{X}}(k, i) = (\_, t, \_)$, then we have that $i \in u\,(k)$.

Now suppose that $i \in u\,(k)$, and let $\mathcal{K}_{\mathcal{X}}(k, i) = (\_, t, \_)$ for some t. This implies that $(\mathtt{w}, k, \_) \in_{\mathcal{X}} t$. By definition $t \in \mathsf{visTx}(\mathcal{K}_{\mathcal{X}}, u)$, hence $t \in T \cup \mathsf{visTx}(\mathcal{K}_{\mathcal{X}}, u))$. Because $t \in T \cup \mathsf{visTx}(\mathcal{K}_{\mathcal{X}}, u)$, then for any key k' such that $(\mathtt{w}, k', \_) \in_{\mathcal{X}} t$, there exists an index $j \in \mathsf{getView}(\mathcal{X}, T \cup \mathsf{visTx}(\mathcal{K}_{\mathcal{X}}, u))$ $\mathcal{K}(k', j) = (\_, t, \_)$; because kv-stores only allow a transaction to write at most one version per key, then the index j is uniquely determined. In particular, we know that $(\mathtt{w}, k, \_) \in_{\mathcal{X}} t$, and $\mathcal{K}_{\mathcal{X}}(k, i) = (\_, t, \_)$, from which it follows that $i \in \mathsf{getView}(\mathcal{X}, T \cup \mathsf{visTx}(\mathcal{K}_{\mathcal{X}}, u))(k)$.

Proposition 9 (Views to visible transactions). *Given a view $u \in \textsc{Views}(\mathcal{K}_{\mathcal{X}})$, there exists $T \subseteq T_{\mathcal{X}}$ such that* $\mathsf{getView}(\mathcal{X}, T) = u$, *and* $\mathsf{RP}_{\mathsf{LWW}}(\mathcal{X}, T) = \mathsf{snapshot}(\mathcal{K}_{\mathcal{X}}, u)$.

Proof. We only need to prove that, for any $u \in \textsc{Views}(\mathcal{K}_{\mathcal{X}})$, there exists $T \subseteq T_{\mathcal{X}}$ such that $\mathsf{getView}(\mathcal{X}, T) = u$. Then it follows from Prop. 7 that $\mathsf{RP}_{\mathsf{LWW}}(\mathcal{X}, T) = \mathsf{snapshot}(\mathcal{K}_{\mathcal{X}}, u)$. It suffices to choose $T = \bigcup_{k \in \textsc{Keys}}(\{\mathsf{w}(\mathcal{K}_{\mathcal{X}}(k, i)) \mid i > 0 \wedge i \in u\,(k)\})$. Fix a key k, and let $i \in u\,(k)$. We prove that $i \in \mathsf{getView}(\mathcal{X}, T)$. If $i = 0$, then $i \in \mathsf{getView}(\mathcal{X}, T)$ by definition. Therefore, assume that $i > 0$. Let $t = \mathsf{w}(\mathcal{K}_{\mathcal{X}}(k, i))$. It must be the case that $t \in T$ and $i \in \mathsf{getView}(\mathcal{X}, T)(k)$.

Next, suppose that $i \in \mathsf{getView}(\mathcal{X}, T)(k)$. We prove that $i \in u\,(k)$. Note that if $i = 0$, then $i \in u\,(k)$ because of the definition of views. Let then $i > 0$. Because

$i \in \mathsf{getView}(\mathcal{X}, T)(k)$, we have that $\mathsf{w}(\mathcal{K}_{\mathcal{X}}(k,i)) \in T$. Let $t = \mathsf{w}(\mathcal{K}_{\mathcal{X}}(k,i))$. Because $i > 0$, it must be the case that $t \neq t_0$. By definition, $t \in T$ only if there exists an index j and key k', possibly different from k, such that $\mathsf{w}(\mathcal{K}_{\mathcal{X}}(k',j)) = t$ and $j \in u\,(k')$. Because $t \neq t_0$ we have that $j > 0$. Finally, because u is atomic by definition, $j \in u\,(k')\ \mathsf{w}(\mathcal{K}_{\mathcal{X}}(k',j)) = t = \mathsf{w}(\mathcal{K}_{\mathcal{X}}(k,i))$, then it must be the case that $i \in u\,(k)$, which concludes the proof.

D.2 KV-Store Traces to Abstract Execution Traces

To prove our definitions using execution test on kv-stores is sound and complete with respect with the axiomatic definitions on abstract executions (§F), we need to prove trace equivalent between these two models.

In this section, we only consider the trace that does not involve `P` but only committing fingerprint and view shift. In §E, we will go further and discuss the trace installed with `P`.

Let $\mathtt{ET}_\top$ be the most permissive execution test. That is $\mathtt{ET}_\top \vdash (\mathcal{K}, u) \triangleright \mathcal{F} : (\mathcal{K}', u')$ such that whenever $u\,(k) \neq u'(k)$ then either $(\mathtt{w}, k, \_) \in \mathcal{F}$ or $(\mathtt{r}, k, \_) \in \mathcal{F}$. We will relate $\mathtt{ET}_\top$-traces to abstract executions that satisfy the last write wins resolution policy, i.e. $(\mathsf{RP}_{\mathsf{LWW}}, \emptyset)$.

To bridge $\mathtt{ET}_\top$-traces to abstract executions, The $\mathsf{absExec}\ (\tau)$ function converse the trace of $\mathtt{ET}_\top$ to set of possible abstract executions (Def. 26). In fact, for any trace τ and abstract execution $\mathcal{X} \in \mathsf{absExec}(\tau)$, the last configuration of τ is $(\mathcal{K}_{\mathcal{X}}, \_)$ (Prop. 10). We often use $\mathcal{X}_\tau$ for $\mathcal{X} \in \mathsf{absExec}(\tau)$.

Definition 26. *Given a kv-store $\mathcal{K}$, a view u, an initial abstract execution $\mathcal{X}_0 = ([], \emptyset, \emptyset)$, an abstract execution $\mathcal{X}$, a set of transactions $T \subseteq T_{\mathcal{X}}$, a transaction identifier t and a set of operations $\mathcal{F}$, the* extend *function defined as the follows:*

$$\mathsf{extend}(\mathcal{X}, t, T, \mathcal{F}) \triangleq \begin{cases} \textit{undefined} & \textit{if } t = t_0 \\ \left(\mathscr{T}_{\mathcal{X}} \uplus \{t \mapsto \mathcal{F}\}, \mathsf{VIS}', \mathsf{AR}'\right) & \textit{if } \dagger \end{cases}$$

$$\dagger \equiv t = t^n_{cl} \wedge \mathsf{VIS}' = \mathsf{VIS}_{\mathcal{X}} \uplus \{(t', t) \mid t \in T\} \wedge \mathsf{AR}' = \mathsf{AR}_{\mathcal{X}} \uplus \{(t', t) \mid t' \in T_{\mathcal{X}}\}$$

Given a $\mathtt{ET}_\top$ trace τ, let $\mathsf{lastConf}(\tau)$ be the last configuration appearing in τ. The set of abstract executions $\mathsf{absExec}(\tau)$ is defined as the smallest set such that:

- $\mathcal{X}_0 \in \mathsf{absExec}((\mathcal{K}_0, \mathcal{U}_0))$,
- *if* $\mathcal{X} \in \mathsf{absExec}(\tau)$, *then* $\mathcal{X} \in \mathsf{absExec}\left(\tau \xrightarrow{(cl,\varepsilon)}_{\mathtt{ET}_\top} (\mathcal{K}, \mathcal{U})\right)$,
- *if* $\mathcal{X} \in \mathsf{absExec}(\tau)$, *then* $\mathcal{X} \in \mathsf{absExec}\left(\tau \xrightarrow{(cl,\emptyset)}_{\mathtt{ET}_\top} (\mathcal{K}, \mathcal{U})\right)$,
- *let* $(\mathcal{K}', \mathcal{U}') = \mathsf{lastConf}(\tau)$*; if* $\mathcal{X} \in \mathsf{absExec}(\tau)$, $\mathcal{F} \neq \emptyset$, *and* $T = \mathsf{visTx}(\mathcal{K}, \mathcal{U}'(cl)) \cup T_{\mathsf{rd}}$ *where* T_{rd} *is a set of* read-only transactions *such that* $(\mathtt{w}, k, v) \notin_{\mathcal{X}} t'$ *for all keys k and values v and transactions $t' \in T_{\mathsf{rd}}$, and if the transaction t is the transaction appearing in* $\mathsf{lastConf}(\tau)$ *but not in $\mathcal{K}$, then* $\mathsf{extend}(\mathcal{X}, t, T, \mathcal{F}) \in \mathsf{absExec}\left(\tau \xrightarrow{(cl,\mathcal{F})}_{\mathtt{ET}_\top} (\mathcal{K}, \mathcal{U})\right)$.

Proposition 10 (Trace of ET to abstract executions). *For any* $\mathtt{ET}_\top$*-trace* τ*, the abstract execution* $\mathcal{X} \in \mathsf{absExec}(\tau)$ *satisfies the last write wins policy, and* $(\mathcal{K}_\mathcal{X}, \_) = \mathsf{lastConf}(\tau)$.

Proof. Fix a $\mathtt{ET}_\top$-trace τ. We prove by induction on the number of transitions n in τ.

- Base case: $n = 0$. It means $\tau = (\mathcal{K}_0, \_)$. It follows from Def. 26 that $\mathcal{X}_\tau = ([], \emptyset, \emptyset)$. This triple satisfies the constraints of Def. 19, as well as the resolution policy $\mathsf{RP}_{\mathsf{LWW}}$. It is also immediate to see that $\mathsf{graphOf}(\mathcal{X}) = ([], \emptyset, \emptyset, \emptyset)$. In particular, $T_{\mathsf{graphOf}(\mathcal{X})} = \emptyset$, and the only kv-store $\mathcal{K}$ such that $T_{\mathcal{G}_\mathcal{K}} = \emptyset$ is given by $\mathcal{K} = \mathcal{K}_0$. By definition, $\mathcal{K}_{\mathcal{X}_\tau} = \mathcal{K}_0$, as we wanted to prove.
- Inductive case: $n > 0$. In this case, we have that $\tau = \tau' \xrightarrow{(cl,\mu)}_{\mathtt{ET}} (\mathcal{K}, \mathcal{U})$ for some $cl, \mu, \mathcal{K}, \mathcal{U}$. The $\mathtt{ET}_\top$-trace τ' contains exactly $n-1$ transitions, so that by induction we can assume that $\mathcal{X}_{\tau'}$ is a valid abstract execution that satisfies $\mathsf{RP}_{\mathsf{LWW}}$. and $\mathsf{lastConf}(\tau') = (\mathcal{K}_{\mathcal{X}_{\tau'}}, \mathcal{U}')$ for some $\mathcal{U}'$.
  We perform a case analysis on μ. If $\mu = \varepsilon$, then it follows that $\mathcal{K} = \mathcal{K}_{\mathcal{X}_{\tau'}}$, and $\mathcal{X}_\tau = \mathcal{X}_{\tau'}$ by Def. 26. Then by the inductive hypothesis $\mathcal{X}_\tau$ is an abstract execution that satisfies $\mathsf{RP}_{\mathsf{LWW}}$, $\mathsf{lastConf}(\tau) = (\mathcal{K}, \_)$, and $\mathcal{K}_{\mathcal{X}_\tau} = \mathcal{K}_{\mathcal{X}_{\tau'}} = \mathcal{K}$, and there is nothing left to prove.
  Suppose now that $\mu = \mathcal{F}$, for some $\mathcal{F}$. In this case we have that $\mathcal{K} \in \mathsf{update}\big(\mathcal{K}_{\mathcal{X}_{\tau'}}, \mathcal{U}'(cl), \mathcal{F}, cl\big)$. Note that if $\mathcal{F} = \emptyset$, then $\mathcal{K} = \mathcal{K}_{\mathcal{X}_{\tau'}}$ and $\mathcal{X}_\tau = \mathcal{X}_{\tau'}$. By the inductive hypothesis, $\mathcal{X}_\tau$ is an abstract execution that satisfies $\mathsf{RP}_{\mathsf{LWW}}$, and $\mathcal{K} = \mathcal{K}_{\mathcal{X}_{\tau'}} = \mathcal{K}_{\mathcal{X}_\tau}$. Assume then that $\mathcal{F} \neq \emptyset$. By definition, $\mathcal{K} = \mathsf{update}\big(\mathcal{K}_{\mathcal{X}_{\tau'}}, \mathcal{U}'(cl), \mathcal{F}, t\big)$ for some $t \in \mathsf{nextTid}(cl, \mathcal{K}_{\mathcal{X}_\tau})$. It follows that t is the unique transaction such that $t \notin \mathcal{K}_{\mathcal{X}_{\tau'}}$, and $t \in \mathcal{K}$ (the fact that $t \in \mathcal{K}$ follows from the assumption that $\mathcal{F} \neq \emptyset$). Let $T = \mathsf{visTx}\big(\mathcal{K}_{\mathcal{X}_{\tau'}}, \mathcal{U}'(cl)\big)$; then $\mathcal{X}_\tau = \mathsf{extend}\big(\mathcal{K}_{\mathcal{X}_{\tau'}}, t, T, \mathcal{F}\big)$. Note that $\mathcal{X}_\tau$ satisfies the constraints of abstract execution required by Def. 19:
  - Because $t \in \mathsf{nextTid}(cl, \mathcal{K}_{\mathcal{X}_\tau})$, it must be the case that $t = t^m_{cl}$ for some $m \geq 1$; we have that $\mathscr{T}_{\mathcal{X}_\tau} = \mathscr{T}_{\mathcal{X}_{\tau'}}[t^m_{cl} \mapsto \mathcal{F}]$, from which it follows that
    $$T_{\mathcal{X}_\tau} = \mathrm{dom}(\mathscr{T}_{\mathcal{X}_\tau}) = \mathrm{dom}(\mathscr{T}_{\mathcal{X}_{\tau'}}) \cup \{t^m_{cl}\} = T_{\mathcal{X}_{\tau'}} \cup \{t^m_{cl}\}$$
    By inductive hypothesis, $t_0 \notin T_{\mathcal{X}_{\tau'}}$, and therefore $t_0 \notin T_{\mathcal{X}_{\tau'}} \cup \{t^m_{cl}\} = T_\mathcal{X}$.
  - $\mathsf{VIS}_{\mathcal{X}_\tau} \subseteq \mathsf{AR}_{\mathcal{X}_\tau}$. Let $(t', t'') \in \mathsf{VIS}_{\mathcal{X}_\tau}$. Then either $t'' = t^m_{cl}$ and $t' \in T$, or $(t', t'') \in \mathsf{VIS}_{\mathcal{X}_{\tau'}}$. In the former case, we have that $(t', t^m_{cl}) \in \mathsf{AR}_{\mathcal{X}_\tau}$ by definition; in the latter case, we have that $(t', t'') \in \mathsf{AR}_{\mathcal{X}_{\tau'}}$ because $\mathcal{X}_{\tau'}$ is a valid abstract execution by inductive hypothesis, and therefore $(t', t'') \in \mathsf{AR}_{\mathcal{X}_\tau}$ by definition. This concludes the proof that $\mathsf{VIS}_{\mathcal{X}_\tau} \subseteq \mathsf{AR}_{\mathcal{X}_\tau}$.
  - $\mathsf{VIS}_{\mathcal{X}_\tau}$ is irreflexive. Assume $(t', t'') \in \mathsf{VIS}_{\mathcal{X}_\tau}$, then either $(t't'') \in \mathsf{VIS}_{\mathcal{X}_{\tau'}}$, and because $\mathsf{VIS}_{\mathcal{X}_{\tau'}}$ is irreflexive by the inductive hypothesis, then $t' \neq t''$; or $t'' = t^m_{cl}$, $t' \in T \subseteq T_{\mathcal{X}_{\tau'}}$, and because $t^m_{cl} \notin \mathcal{K}_{\mathcal{X}_{\tau'}}$, then $t' \neq t^m_{cl}$.
  - $\mathsf{AR}_{\mathcal{X}_\tau}$ is total. Let $(t', t'') \in T_{\mathcal{X}_\tau}$. Suppose that $t' \neq t''$.

(1) If $t' \neq t^m_{cl}$, $t'' \neq t^m_{cl}$, then it must be the case that $t', t'' \in T_{\mathcal{X}_{\tau'}}$; this is because we have already argued that $T_{\mathcal{X}_\tau} = T_{\mathcal{X}_{\tau'}} \cup \{t^m_{cl}\}$. By the inductive hypothesis, we have that either $(t', t'') \in \mathsf{AR}_{\mathcal{X}_{\tau'}}$, or $(t'', t') \in \mathsf{AR}_{\mathcal{X}_{\tau'}}$. Because $\mathsf{AR}_{\mathcal{X}_{\tau'}} \subseteq \mathsf{AR}_{\mathcal{X}_\tau}$, then either $(t', t'') \in \mathsf{AR}_{\mathcal{X}_{\tau'}}$ or $(t'', t') \in \mathsf{AR}_{\mathcal{X}_\tau}$.
(2) if $t'' = t^m_{cl}$, then it must be $t' \in T_{\mathcal{X}_{\tau'}}$. By definition, $(t', t^m_{cl}) \in \mathsf{AR}_{\mathcal{X}_\tau}$. Similarly, if $t' = t^m_{cl}$, we can prove that $(t'', t^m_{cl}) \in \mathsf{AR}_{\mathcal{X}_\tau}$.

- $\mathsf{AR}_{\mathcal{X}_\tau}$ is irreflexive. It follows is the same as the one of $\mathsf{VIS}_{\mathcal{X}_\tau}$.
- $\mathsf{AR}_{\mathcal{X}_\tau}$ is transitive. Assume $(t', t'') \in \mathsf{AR}_{\mathcal{X}_\tau}$ and $(t'', t''') \in \mathsf{AR}_{\mathcal{X}_\tau}$. Note that it must be the case that $t', t'' \in T_{\mathcal{X}_{\tau'}}$ by the definition of $\mathsf{AR}_{\mathcal{X}}$, and in particular $(t', t'') \in \mathsf{AR}_{\mathcal{X}_{\tau'}}$. For t''', we have two possible cases.
  (1) Either $t''' \in T_{\mathcal{X}_\tau}$, from which it follows that $(t'', t''') \in \mathsf{AR}_{\mathcal{X}_{\tau'}}$; because of $\mathsf{AR}_{\mathcal{X}_{\tau'}}$ is transitive by the inductive hypothesis, then $(t', t''') \in \mathsf{AR}_{\mathcal{X}_{\tau'}}$, and therefore $(t', t''') \in \mathsf{AR}_{\mathcal{X}_\tau}$.
  (2) Or $t''' = t^m_{cl}$, and because $t' \in T_{\mathcal{X}_{\tau'}}$, then $(t', t^m_{cl}) \in \mathsf{AR}_{\mathcal{X}_\tau}$ by definition.
- $\mathsf{SO}_{\mathcal{X}_\tau} \subseteq \mathsf{AR}_{\mathcal{X}_\tau}$. Let cl' be a client such that $(t^i_{cl'}, t^j_{cl'}) \in \mathsf{AR}_{\mathcal{X}_\tau}$. If $cl' \neq cl$, then it must be the case that $t^i_{cl'}, t^j_{cl'} \in T_{\mathcal{X}_{\tau'}}$, and therefore $(t^i_{cl'}, t^j_{cl'}) \in \mathsf{AR}_{\mathcal{X}_{\tau'}}$. By the inductive hypothesis, it follows that $i < j$. If $cl' = cl$, then by definition of $\mathsf{AR}_{\mathcal{X}_\tau}$ it must be $i \neq m$. If $j \neq m$ we can proceed as in the previous case to prove that $i < j$. If $j = m$, then note that $t^i_{cl} \in T_{\mathcal{X}_\tau}$ only if $t^i_{cl} \in \mathcal{K}_{\mathcal{X}_{\tau'}}$. Because $t^m_{cl} \in \mathsf{nextTid}(\mathcal{K}_{\mathcal{X}_{\tau'}}, cl)$, then we have that $i < m$, as we wanted to prove.

Next, we prove that $\mathcal{X}_\tau$ satisfies the last write wins policy. Let $t' \in T_{\mathcal{X}_\tau}$, and suppose that $(\mathtt{r}, k, v) \in_{\mathcal{X}_\tau} t'$.

- If $t' \neq t$, then we have that $t \in T_{\mathcal{X}_{\tau'}}$. We also have that $\mathsf{VIS}^{-1}_{\mathcal{X}_\tau}(t') = \mathsf{VIS}^{-1}_{\mathcal{X}_{\tau'}}(t')$, $\mathsf{AR}^{-1}_{\mathcal{X}_\tau}(t') = \mathsf{AR}^{-1}_{\mathcal{X}_{\tau'}}(t')$; finally, for any $t'' \in T_{\mathcal{X}_{\tau'}}$, $(\mathtt{w}, k, v') \in_{\mathcal{X}_\tau} t''$ if and only if $(\mathtt{w}, k, v') \in_{\mathcal{X}_{\tau'}} t''$. Therefore, let $t_r = \max_{\mathsf{AR}_{\mathcal{X}_\tau}}(\mathsf{VIS}^{-1}_{\mathcal{X}_\tau}(t') \cap \{t'' \mid (\mathtt{w}, k, \_) \in_{\mathcal{X}_\tau} t''\})$. We have that

$$t_r = \max_{\mathsf{AR}_{\mathcal{X}_{\tau'}}} (\mathsf{VIS}^{-1}_{\mathcal{X}_{\tau'}}(t) \cap \left\{t'' \;\middle|\; (\mathtt{w}, k, \_) \in_{\mathcal{X}_{\tau'}} t''\right\})$$

  and because $\mathcal{X}_{\tau'}$ satisfies the last write wins resolution policy, then $(\mathtt{w}, k, v) \in_{\mathcal{X}_{\tau'}} t_r$. This also implies that $(\mathtt{w}, k, v) \in_{\mathcal{X}_\tau} t_r$.
- Now, suppose that $t' = t$. Suppose that $(\mathtt{r}, k, v) \in_{\mathcal{X}_\tau} t'$. By definition, we have that $(\mathtt{r}, k, v) \in \mathcal{F}$. Recall that $\tau = \tau' \xrightarrow{(cl,\mathcal{F})}_{\mathtt{ET}_\top} (\mathcal{K}, \mathcal{U})$, and $\mathsf{lastConf}(\tau') = (\mathcal{K}_{\mathcal{X}_{\tau'}}, \mathcal{U}')$ for some $\mathcal{U}'$. That is,

$$(\mathcal{K}_{\mathcal{X}_{\tau'}}, \mathcal{U}') \xrightarrow{(cl,\mathcal{F})}_{\mathtt{ET}_\top} (\mathcal{K}, \mathcal{U})$$

  which in turn implies that $\mathtt{ET}_\top \vdash (\mathcal{K}_{\mathcal{X}_{\tau'}}, \mathcal{U}'(cl)) \triangleright \mathcal{F} : (\mathcal{K}, \mathcal{U}(cl))$. Let then $r = \max\{i \mid i \in \mathcal{U}'(cl)(k)\}$. By definition of execution test, and because $(\mathtt{r}, k, v) \in \mathcal{F}$, then it must be the case that $\mathcal{K}_{\mathcal{X}_{\tau'}}(k, r) = (v, t'', \_)$ for some t''.

We now prove that $t'' = \max_{\mathsf{AR}_{\mathcal{X}_\tau}}(\mathsf{VIS}^{-1}_{\mathcal{X}_\tau}(t) \cap \{t'' \mid (\mathtt{w}, k, \_) \in_{\mathcal{X}_\tau} t''\})$. First we have

$$\mathsf{VIS}^{-1}_{\mathcal{X}_\tau}(t) = \mathsf{visTx}\big(\mathcal{K}_{\mathcal{X}_{\tau'}}, \mathcal{U}'(cl)\big) = \big\{\mathsf{w}(\mathcal{K}_{\mathcal{X}_{\tau'}}(k', i)) \;\big|\; k' \in \textsc{Keys} \wedge i \in \mathcal{U}'(cl)(k')\big\}$$

Note that $r \in \mathcal{U}'(cl)(k)$, and $t'' = \mathsf{w}(\mathcal{K}_{\mathcal{X}_{\tau'}}(k, r))$. Therefore, $t'' \in \mathsf{VIS}^{-1}_{\mathcal{X}_\tau}(t)$. Because $\mathcal{K} = \mathsf{update}\big(\mathcal{K}_{\mathcal{X}_{\tau'}}, \mathcal{U}'(cl), \mathcal{F}, t\big)$, it must be the case that $\mathsf{w}(\mathcal{K}(k, r)) = t''$. Also, because $\mathsf{w}(\mathcal{K}_{\mathcal{X}_{\tau'}}(k, r)) = t''$, then $(\mathtt{w}, k, \_) \in_{\mathcal{X}_{\tau''}} t''$, or equivalently $(\mathtt{w}, k, \_) \in \mathscr{T}_{\mathcal{X}_{\tau'}}(t'')$. We have already proved that $\mathsf{VIS}_{\mathcal{X}_\tau}$ is irreflexive, hence it must be the case that $t'' \neq t$. In particular, because $\mathcal{X}_\tau = \mathsf{extend}(\mathcal{X}_{\tau'}, t, \_, \_)$, then we have that $\mathscr{T}_{\mathcal{X}_\tau}(t'') = \mathscr{T}_{\mathcal{X}_{\tau'}}[t \mapsto \mathcal{F}](t'') = \mathscr{T}_{\mathcal{X}_{\tau'}}(t'')$, hence $(\mathtt{w}, k, \_) \in \mathscr{T}_{\mathcal{X}_\tau}(t'')$. Equivalently, $(\mathtt{w}, k, \_) \in_{\mathcal{X}_\tau} t''$. We have proved that $t'' \in \mathsf{VIS}^{-1}_{\mathcal{X}_\tau}(t)$, and $(\mathtt{w}, k, \_) \in_{\mathcal{X}_\tau} t''$.
Now let t''' be such that $t''' \in \mathsf{VIS}^{-1}_{\mathcal{X}_\tau}(t)$, and $(\mathtt{w}, k\_) \in_{\mathcal{X}_\tau} t'''$. Note that $t''' \neq t$ because $\mathsf{VIS}_{\mathcal{X}_\tau}$ is irreflexive. We show that either $t''' = t''$, or $t''' \xrightarrow{\mathsf{AR}_{\mathcal{X}_\tau}} t''$. Because $t''' \in \mathsf{VIS}^{-1}_{\mathcal{X}_\tau}(t)$, then there exists a key k' and an index $i \in \mathcal{U}'(cl)$ such that $\mathsf{w}(\mathcal{K}_{\mathcal{X}_{\tau'}}(k', i)) = t'''$. Because $(\mathtt{w}, k, \_) \in_{\mathcal{X}_\tau} t'''$, and because $t''' \neq t$, then $(\mathtt{w}, k, \_) \in_{\mathcal{X}_{\tau'}} t'''$, and therefore there exists an index j such that $\mathsf{w}(\mathcal{K}_{\mathcal{X}_{\tau'}}(k, j)) = t'''$. We have that $\mathsf{w}(\mathcal{K}_{\mathcal{X}_{\tau'}}(k, j) = \mathsf{w}(\mathcal{K}_{\mathcal{X}_{\tau'}}(k', i))$, and $i \in \mathcal{U}'(cl)$. By Eq. (atomic), it must be $j \in \mathcal{U}'(cl)$. Note that $r = \max\{i \mid i \in \mathcal{U}'(cl)\}$, hence we have that $j \leq r$. If $j = r$, then $t''' = t''$ and there is nothing left to prove. If $j < r$, then we have that $(t''', t'') \in \mathsf{AR}_{\mathcal{X}_{\tau'}}$, and therefore $(t''', t'') \in \mathsf{AR}_{\mathcal{X}_\tau}$.

Finally, we need to prove that $\mathcal{K} = \mathcal{K}_{\mathcal{X}_\tau}$. Recall $\mathcal{K} = \mathsf{update}\big(\mathcal{K}_{\mathcal{X}_{\tau'}}, \mathcal{U}'(cl), \mathcal{F}, t\big)$, and $\mathcal{X}_\tau = \mathsf{extend}\big(\mathcal{X}_{\tau'}, \mathsf{visTx}\big(\mathcal{K}_{\mathcal{X}_{\tau'}}, \mathcal{U}'(cl)\big), t, \mathcal{F}\big)$. The result follows then from Prop. 11.

Proposition 11 (extend **matching** update)**.** *Given an abstract execution $\mathcal{X}$, a set of transactions $T \subseteq T_{\mathcal{X}}$, a transaction $t \notin T_{\mathcal{X}}$, and a fingerprint $\mathcal{F} \subseteq \mathcal{P}(\textsc{Ops})$, if the new abstract execution $\mathcal{X}' = \mathsf{extend}(\mathcal{X}, T, t, \mathcal{F})$, and the view $u = \mathsf{getView}(\mathcal{K}_{\mathcal{X}}, T)$, then $\mathsf{update}(\mathcal{K}_{\mathcal{X}}, u, \mathcal{F}, t) = \mathcal{K}_{\mathcal{X}'}$.*

Proof. Let $\mathcal{G} = \mathcal{G}_{\mathsf{update}(\mathcal{K}_{\mathcal{X}}, u, \mathcal{F}, t)}$, $\mathcal{G}' = \mathsf{graphOf}(\mathcal{X}')$. Note that $\mathcal{K}_{\mathcal{X}'}$ is the unique kv-store such that $\mathcal{G}_{\mathcal{K}_{\mathcal{X}'}} = \mathsf{graphOf}(\mathcal{X}') = \mathcal{G}'$. It suffices to prove that $\mathcal{G} = \mathcal{G}'$. Because the function $\mathcal{G}.$ is injective, it follows that $\mathsf{update}(\mathcal{K}_{\mathcal{X}}, u, \mathcal{F}, t) = \mathcal{K}_{\mathcal{X}'}$, as we wanted to prove.

The proof is a consequence of Lemma 1 and Lemma 2. Consider the dependency graph $\mathcal{G}_{\mathcal{K}_{\mathcal{X}}}$. Recall that $\mathcal{K}_{\mathcal{X}}$ is the unique kv-store such that $\mathcal{G}_{\mathcal{K}_{\mathcal{X}}} = \mathsf{graphOf}(\mathcal{X})$. We prove that $\mathscr{T}_{\mathcal{G}} = \mathscr{T}_{\mathcal{G}'}$, $\mathsf{WR}_{\mathcal{G}} = \mathsf{WR}_{\mathcal{G}'}$ and $\mathsf{WW}_{\mathcal{G}} = \mathsf{WW}_{\mathcal{G}'}$ (from the last two it follows that $\mathsf{RW}_{\mathcal{G}} = \mathsf{RW}_{\mathcal{G}'}$).

- It is easy to see $\mathscr{T}_{\mathcal{G}} = \mathscr{T}_{\mathcal{G}'}$.
- $\mathsf{WR}_{\mathcal{G}} = \mathsf{WR}_{\mathcal{G}'}$. Let $\mathcal{K} = \mathcal{K}_{\mathcal{X}}$. Suppose that $t' \xrightarrow{\mathsf{WR}_{\mathcal{G}}(k)} t''$ for some t', t''. By Lemma 2 we have that either $t' \xrightarrow{\mathsf{WR}_{\mathcal{G}_{\mathcal{K}}}(k)} t''$, or $t'' = t$, $(\mathtt{r}, k, \_) \in \mathcal{F}$, $t' = \max_{\mathsf{WW}_{\mathcal{G}_{\mathcal{K}}}(k)}\{\mathsf{w}(k, i) \mid i \in u\,(k)\}$.

- If $t' \xrightarrow{\mathsf{WR}_{\mathcal{G}_\mathcal{K}}(k)} t''$, then because $\mathcal{G}_\mathcal{K} = \mathsf{graphOf}(\mathcal{X})$, we have that $t' \xrightarrow{\mathsf{WR}_{\mathsf{graphOf}(\mathcal{X})}(k)} t''$. Recall that $\mathcal{G}' = \mathsf{graphOf}(\mathsf{extend}(\mathcal{X}, T, t, \mathcal{F}))$, hence by Lemma 1 we obtain that $t' \xrightarrow{\mathsf{WR}_{\mathcal{G}'}(k)} t''$.
- If $t'' = t$, $(\mathtt{r}, k, \_) \in \mathcal{F}$, and $t' = \max_{\mathsf{WW}_{\mathcal{G}_\mathcal{K}}(k)} \{\mathsf{w}(\mathcal{K}_\mathcal{X}(k,i)) \mid i \in u(k)\}$, then we also have that $t' = \max_{\mathsf{WW}_{\mathsf{graphOf}(\mathcal{X})}(k)}(T \cap \{t''' \mid (\mathtt{w}, k, \_) \in_\mathcal{X} t'''\})$. This is because of the assumption that

$$\begin{aligned}\{\mathsf{w}(\mathcal{K}_\mathcal{X}(k,i)) \mid i \in u(k)\} &= \{\mathsf{w}(\mathcal{K}_\mathcal{X}(k',i)) \mid k' \in \textsc{Keys} \wedge i \in u(k')\} \cap \{\mathsf{w}(\mathcal{K}_\mathcal{X}(k,\_)\} \\ &= \mathsf{visTx}(\mathcal{K}_\mathcal{X}, u) \cap \{\mathsf{w}(\mathcal{K}_\mathcal{X}(k,\_)\} \\ &= T \cap \{t''' \mid (\mathtt{w}, k, \_) \in_\mathcal{X} t'''\}\end{aligned}$$

  Again, it follows from Lemma 1 that $t' \xrightarrow{\mathsf{WR}_{\mathcal{G}'}(k)} t''$.

– $\mathsf{WW}_\mathcal{G} = \mathsf{WW}_{\mathcal{G}'}$. The $\mathsf{WW}_\mathcal{G} = \mathsf{WW}_{\mathcal{G}'}$ follows the similar reasons as $\mathsf{WR}_\mathcal{G} = \mathsf{WR}_{\mathcal{G}'}$.

Lemma 1 (Graph to abstract execution extension). *Let $\mathcal{X}$ be an abstract execution, $t \notin T_\mathcal{X} \cup \{t_0\}$ be a transaction identifier $T_\mathcal{X}$, and $\mathcal{F} \in T_\mathcal{X}$. Let $T \subseteq T_\mathcal{X}$ be a set of transaction identifiers. Let $\mathcal{G} = \mathsf{graphOf}(\mathcal{X}), \mathcal{G}' = \mathsf{graphOf}(\mathsf{extend}(\mathcal{X}, t, T, \mathcal{F}))$. We have the following:*

(1) for any $t' \in T_{\mathcal{G}'}$, either $t' \in T_\mathcal{G}$ and $\mathscr{T}_\mathcal{G}(t') = \mathscr{T}_{\mathcal{G}'}(t')$, or $t' = t$ and $\mathscr{T}_{\mathcal{G}'}(t) = \mathcal{F}$.

(2) $t' \xrightarrow{\mathsf{WR}_{\mathcal{G}'}(k)} t''$ if and only if either $t' \xrightarrow{\mathsf{WR}_\mathcal{G}(k)_\mathcal{G}} t''$, or $(\mathtt{r}, k, \_) \in \mathcal{F}$, $t'' = t$ and $t' = \max_{\mathsf{WW}_\mathcal{G}(k)}(T)$,

(3) $t' \xrightarrow{\mathsf{WW}_{\mathcal{G}'}(k)} t''$ if and only if either $t' \xrightarrow{\mathsf{WW}_\mathcal{G}(k)} t''$, or $(\mathtt{w}, k, \_) \in \mathcal{F}$, $t'' = t$, and $(\mathtt{w}, k, \_) \in_\mathcal{G} t'$.

Proof. Fix a key k. Let $\mathcal{X}' = \mathsf{extend}(\mathcal{X}, t, T, \mathcal{F})$. Recall that $\mathcal{G}' = \mathsf{graphOf}(\mathcal{X}')$.

(1) By definition of extend, and because $t \notin T_\mathcal{X}$, we have that $T_{\mathcal{X}'} = T_\mathcal{X} \uplus \{t\}$. Furthermore, $\mathscr{T}_{\mathcal{X}'}(t) = \mathcal{F}$, from which it follows that $\mathscr{T}_{\mathcal{G}'}(t) = \mathcal{F}$. For all $t' \in T_\mathcal{X}$, we have that $\mathscr{T}_{\mathcal{X}'}(t') = \mathscr{T}_\mathcal{X}(t') = \mathscr{T}_\mathcal{G}(t')$.

(2) There are two cases that either the t'' already exists in the dependency graph before, or it is the newly committed transaction.

– Suppose that $t' \xrightarrow{\mathsf{WR}(k)_\mathcal{G}} t''$ for some $t', t'' \in T_\mathcal{G}$. By definition, $(\mathtt{r}, k, \_) \in_\mathcal{X} t''$, and $t' = \max_{\mathsf{AR}_\mathcal{X}}(\mathsf{VIS}^{-1}_\mathcal{X}(t'') \cap \{t''' \mid (\mathtt{w}, k, \_) \in_\mathcal{X} t'''\})$. Because $t'' \in T_\mathcal{G} = T_\mathcal{X}$, it follows that $t'' \neq t$. By definition, $\mathsf{VIS}^{-1}_{\mathcal{X}'}(t'') = \mathsf{VIS}^{-1}_\mathcal{X}(t)$: also, whenever $t_a, t_b \in \mathsf{VIS}^{-1}_{\mathcal{X}'}(t)$ we have that $t_a, t_b \in T_\mathcal{X}$, and therefore if $t_a \xrightarrow{\mathsf{AR}_{\mathcal{X}'}} t_b$, then it must be the case that $t_a \xrightarrow{\mathsf{AR}_\mathcal{X}} t_b$; also, $\mathscr{T}_\mathcal{X}(t_a) = \mathscr{T}_{\mathcal{X}'}(t_a)$. As a consequence, we have that

$$\max{}_{\mathsf{AR}_{\mathcal{X}'}}(\mathsf{VIS}^{-1}_{\mathcal{X}'}(t) \cap \{t''' \mid (\mathtt{w}, k, \_) \in_{\mathcal{X}'} t'''\}) = \max{}_{\mathsf{AR}_\mathcal{X}}(\mathsf{VIS}^{-1}_\mathcal{X}(t) \cap \{t''' \mid (\mathtt{w}, k, \_) \in_\mathcal{X} t'''\}) = t'$$

and therefore $t' \xrightarrow{\mathsf{WR}_{\mathcal{G}'}} t$.

– Suppose now that $(\mathtt{r}, k, \_) \in \mathcal{F}$, and $t' = \max_{\mathsf{WW}(k)_{\mathcal{G}}}(T)$. By Definition, $t' = \max_{\mathsf{AR}_{\mathcal{X}}}(T) \cap \{t''' \mid (\mathtt{w}, k, \_) \in_{\mathcal{X}} t'''\}$, and, $T = \mathsf{VIS}^{-1}_{\mathcal{X}'}(t)$. Because $T \subseteq T_{\mathcal{X}}$, we have that for any t_a, t_b, if $t_a \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t_b$, then $t_a \xrightarrow{\mathsf{AR}_{\mathcal{X}'}} t_b$; and $\mathscr{T}_{\mathcal{X}'}(t_a) = \mathscr{T}_{\mathcal{X}}(t_a)$. Therefore,

$$t' = \max{}_{\mathsf{AR}_{\mathcal{X}'}}(\mathsf{VIS}^{-1}_{\mathcal{X}'}(t) \cap \{t''' \mid (\mathtt{w}, k, \_) \in_{\mathcal{X}'} t'''\}\,,$$

from which it follows that $t' \xrightarrow{\mathsf{WR}_{\mathcal{G}'}(k)} t$.
Now, suppose that $t' \xrightarrow{\mathsf{WR}_{\mathcal{G}'}(k)} t''$ for some $t', t'' \in T_{\mathcal{G}'} = T_{\mathcal{X}'}$. We have that $(\mathtt{r}, k, \_) \in_{\mathcal{X}'} t''$, $(\mathtt{w}, k, \_) \in_{\mathcal{X}'} t'$, and $t'' = \max_{\mathsf{AR}_{\mathcal{X}'}}(\mathsf{VIS}^{-1}_{\mathcal{X}'}(t'') \cap \{t''' \mid (\mathtt{w}, k, \_) \in_{\mathcal{X}'} t'''\}$. We also have that $T_{\mathcal{X}'} = T_{\mathcal{X}} \uplus \{t\}$. We perform a case analysis on t''.

- If $t'' = t$, then by definition of extend we have that $\mathsf{VIS}^{-1}_{\mathcal{X}'}(t) = T$. Note that $T \subseteq T_{\mathcal{X}}$, so that for any $t_a, t_b \in T_{\mathcal{X}}$, we have that $t_a \xrightarrow{\mathsf{AR}_{\mathcal{X}'}} t_b$ if and only if $t_a \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t_b$, and $(\mathtt{w}, k, v) \in_{\mathcal{X}'} t_a$ if and only if $(\mathtt{w}, k, v) \in_{\mathcal{X}} t_a$. Thus, $t' = \max_{\mathsf{AR}_{\mathcal{X}}}(T \cap \{t''' \mid (\mathtt{w}, k, \_) \in_{\mathcal{X}} t'''\}) = \max_{\mathsf{WW}_{\mathcal{G}}(k)}(T)$.
- If $t'' \in T_{\mathcal{X}}$, then it is the case that $t' = \max_{\mathsf{AR}_{\mathcal{X}'}}(\mathsf{VIS}^{-1}_{\mathcal{X}'}(t'') \cap \{t''' \mid (\mathtt{w}, k, \_) \in_{\mathcal{X}'} t'''\}$. Similarly to the case above, we can prove that $\mathsf{VIS}^{-1}_{\mathcal{X}'}(t'') = \mathsf{VIS}^{-1}_{\mathcal{X}}(t)$, for any $t_a, t_b \in \mathsf{VIS}^{-1}_{\mathcal{X}}(t)$, $(\mathtt{w}, k, v) \in_{\mathcal{X}'} t_a$ implies $(\mathtt{w}, k, v) \in_{\mathcal{X}} t_a$, and $t_a \xrightarrow{\mathsf{AR}_{\mathcal{X}'}} t_b$ implies $t_a \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t_b$, from which it follows that $t' = \max_{\mathsf{AR}_{\mathcal{X}}}(\mathsf{VIS}^{-1}_{\mathcal{X}}(t'') \cap \{t''' \mid (\mathtt{w}, k\_) \in_{\mathcal{X}} t'''\})$, and therefore $t' \xrightarrow{\mathsf{WR}_{\mathcal{G}}(k)} t''$.

(3) Similar to $\mathsf{WR}\,(k)_{\mathcal{G}}$, there are two cases that either the t'' already exists in the dependency graph before, or it is the newly committed transaction.

– Suppose that $t' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(k)} t''$ for some $t', t'' \in T_{\mathcal{X}}$. Then $(\mathtt{w}, k, \_) \in_{\mathcal{X}} t', (\mathtt{w}, k, \_) \in_{\mathcal{X}} t''$, and $t' \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t''$. By definition of extend, it follows that $t' \xrightarrow{\mathsf{AR}_{\mathcal{X}'}} t''$, and because $t', t'' \in T_{\mathcal{X}}$, hence $t', t'' \neq t$, then $(\mathtt{w}, k, \_) \in_{\mathcal{X}'} t'$, $(\mathtt{w}, k, \_) \in_{\mathcal{X}'} t''$. By definition, we have that $t' \xrightarrow{\mathsf{WW}_{\mathcal{X}'}(k)} t''$.
– Suppose that $(\mathtt{w}, k, \_) \in_{\mathcal{X}} t'$, $(\mathtt{w}, k, \_) \in \mathcal{F}$. Because $t' \in T_{\mathcal{X}}$, we have that $t' \neq t$, hence $(\mathtt{w}, k, \_) \in_{\mathcal{X}'} t'$. By definition, $\mathscr{T}_{\mathcal{X}'}(t) = \mathcal{F}$, hence $(\mathtt{w}, k, \_) \in_{\mathcal{X}'} t$. Finally, the definition of extend ensures that $t' \xrightarrow{\mathsf{AR}_{\mathcal{X}'}} t$. Combining these three facts together, we obtain that $t' \xrightarrow{\mathsf{WW}_{\mathcal{G}'}(k)} t$.
Now, suppose that $t' \xrightarrow{\mathsf{WW}_{\mathcal{G}'}(k)} t''$ for some $t', t'' \in T_{\mathcal{X}}$. Then $t' \xrightarrow{\mathsf{AR}_{\mathcal{X}'}} t''$, $(\mathtt{w}, k, \_) \in_{\mathcal{X}'} t'$, $(\mathtt{w}, k, \_) \in_{\mathcal{X}'} t''$. Recall that $T_{\mathcal{G}'} = T_{\mathcal{X}'} = T_{\mathcal{X}} \uplus \{t\}$. We perform a case analysis on t''.

- If $t'' = t$, then the definition of extend ensures that $t' \xrightarrow{\mathsf{AR}_{\mathcal{X}'}} t$ implies that $t \in T_{\mathcal{X}}$, hence $t' \neq t$. Together with $(\mathtt{w}, k, \_) \in_{\mathcal{X}'} t'$, this leads to $(\mathtt{w}, k, \_) \in_{\mathcal{X}} t'$.
- If $t'' \in T_{\mathcal{X}}$, then $t'' \neq t$. The definition of extend ensures that $t' \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t''$. This implies that $t', t'' \in T_{\mathcal{X}}$, hence $t', t'' \neq t$, and $\mathscr{T}_{\mathcal{X}'}(t') =$

$\mathscr{T}_{\mathcal{X}}(t')$, $\mathscr{T}_{\mathcal{X}'}(t'') = \mathscr{T}_{\mathcal{X}}(t'')$. It follows that $(\mathtt{w}, k, \_) \in_{\mathcal{X}} t'$, $(\mathtt{w}, k, \_) \in_{\mathcal{X}} t''$, and therefore $t' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(k)} t''$.

Lemma 2 (Graph to kv-store update). *Let $\mathcal{K}$ be a kv-store, and $u \in \textsc{Views}(\mathcal{K})$. Let $t \notin \mathcal{K}$, and $\mathcal{F} \subseteq \mathcal{P}(\textsc{Ops})$, and let $\mathcal{K}' = \mathsf{update}(\mathcal{K}, u, \mathcal{F}, t)$. Let $\mathcal{G} = \mathcal{G}_{\mathcal{K}}$, $\mathcal{G}' = \mathcal{G}_{\mathcal{K}'}$; then for all $t', t'' \in T_{\mathcal{G}'}$ and keys k,*

- $\mathscr{T}_{\mathcal{G}'} = \mathscr{T}_{\mathcal{G}}[t \mapsto \mathcal{F}]$,
- $t' \xrightarrow{\mathsf{WR}_{\mathcal{G}'}(k)} t''$ *if and only if either* $t' \xrightarrow{\mathsf{WR}_{\mathcal{G}}(k)} t''$, *or* $(\mathtt{r}, k, \_) \in \mathcal{F}$ *and*

$$t' = \max_{\mathsf{WW}_{\mathcal{G}}(k)}(\{\mathsf{w}(\mathcal{K}(k,i))\}\, i \in u\,(k))$$

- $t' \xrightarrow{\mathsf{WW}_{\mathcal{G}'}(k)} t''$ *if and only if either* $t' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(k)} t''$, *or* $(\mathtt{w}, k, \_) \in \mathcal{F}$ *and* $t' = \mathsf{w}(\mathcal{K}(k, \_))$.

Proof. Fix $k \in \textsc{Keys}$. Because $t \notin \mathcal{K}$, then $t \notin T_{\mathcal{G}}$, and by definition of *update* we obtain that $\{t' \mid t' \in \mathcal{K}'\} = \{t' \mid t' \in \mathcal{K}\} \cup \{t\}$. It follows that $T_{\mathcal{G}'} = T_{\mathcal{G}} \uplus \{t\}$.

(1) Suppose that $(\mathtt{r}, k, v) \in_{\mathcal{G}} t'$. By definition, there exists an index i such that $\mathcal{K}(k,i) = (v, \_, \{t'\} \cup \_)$. Because $\mathcal{K}' = \mathsf{update}(\mathcal{K}, u, \mathcal{F}, t)$, it is immediate to observe that $\mathcal{K}'(k,i) = (v, \_, \{t'\} \cup \_)$, and therefore $(\mathtt{r}, k, v) \in_{\mathcal{G}'} t'$. Conversely, note that if $(\mathtt{r}, k, v) \in_{\mathcal{G}'} t$, then there exists an index $i = 0, \cdots, |\mathcal{K}'(k)| - 1$ such that $\mathcal{K}'(k,i) = (v, \_, \{t'\} \cup \_)$. it follows that it must be the case that $i \leq |\mathcal{K}(k)| - 1$, and because $t' \neq t$, we have that $\mathcal{K}(k,i) = (v, \_, \{t'\} \cup \_)$. Therefore $(\mathtt{r}, k, v) \in_{\mathcal{G}} t'$.
Similarly, if $(\mathtt{w}, k, v) \in_{\mathcal{G}} t'$, then there exists an index $i = 0, \cdots, |\mathcal{K}(k)| - 1$ such that $\mathcal{K}(k,i) = (v, t', v)$. It follows that $\mathcal{K}'(k,i) = (v, t', \_)$, hence $(\mathtt{w}, k, v) \in_{\mathcal{G}'} t'$. If $(\mathtt{w}, k, v) \in \mathcal{F}$, then we have that $\mathcal{K}'(k, |\mathcal{K}'(k)|-1) = (v, t', \_)$, hence $(\mathtt{w}, k, v) \in_{\mathcal{G}'} t'$. Conversely, if $(\mathtt{w}, k, v) \in_{\mathcal{G}'} t'$, then there exists an index $i = 0, \cdots, |\mathcal{K}'(k)| - 1$ such that $\mathcal{K}(k,i) = (v, t', \_)$. We have two possible cases: either $i < |\mathcal{K}'(k,i)| - 1$, leading to $t' \neq t$ and $\mathcal{K}(k,i) = (v, t', \_)$, or equivalently $(\mathtt{r}, k, v) \in_{\mathcal{G}} t'$; or $i = |\mathcal{K}'(k,i)| - 1$, leading to $t' = t$, and $\mathcal{K}(k,i) = (v, t, \emptyset)$ for some v such that $(\mathtt{w}, k, v) \in \mathcal{F}$.
Putting together the facts above, we obtain that $\mathscr{T}_{\mathcal{G}'} = \mathscr{T}_{\mathcal{G}}[t \mapsto \mathcal{F}]$, as we wanted to prove.

(2) There are two cases that either the t'' already exists in the dependency graph before, or it is the newly committed transaction.
   - Suppose that $t' \xrightarrow{\mathsf{WR}_{\mathcal{G}}(k)} t''$. By definition, there exists an index $i = 0, \cdots, |\mathcal{K}(k)| - 1$ such that $\mathcal{K}(k,i) = (\_, t', \{t''\} \cup \_)$. It is immediate to observer, from the definition of update, that $\mathcal{K}'(k,i) = (\_, t', \{t''\} \cup \_)$, and therefore $t' \xrightarrow{\mathsf{WR}_{\mathcal{G}'}(k)} t''$.
   - Next, suppose that $(\mathtt{r}, k, \_) \in \mathcal{F}$, and $t' = \max_{\mathsf{WW}_{\mathcal{G}}(k)}(\{\mathsf{w}(\mathcal{K}(k,i)) \mid i \in u\,(k)\}$. By Definition, $\mathcal{K}(k,i) = (\_, t', \_)$, where $i = \max(u\,(k))$. This is because $t' \to \mathsf{WW}_{\mathcal{G}}(k) t''$ if and only if $t' = \mathsf{w}(\mathcal{K}(k, j_1)), t'' = \mathsf{w}(\mathcal{K}(k, j_2))$ for

some j_1, j_2 such that $j_1 < j_2$. The definition of update now ensures that $\mathcal{K}'(k,i) = (\_, t', \{t\} \cup \_)$, from which it follows that $t' \xrightarrow{\mathsf{WR}_{\mathcal{G}'}(k)} t$.

Conversely, suppose that $t' \xrightarrow{\mathsf{WR}_{\mathcal{G}'}(k)} t''$. Recall that $T_{\mathcal{G}'} = T_{\mathcal{G}} \cup \{t\}$, hence either $t'' \in T_{\mathcal{G}}$ or $t'' = t$.

- If $t'' = t$, then it must be the case that there exists an index $i = 0, \cdots, |\mathcal{K}'(k)| - 1$ such that $\mathcal{K}'(k,i) = (\_, t', \{t\} \cup \_)$. Note that if $\mathcal{K}'(k, |\mathcal{K}'(k)|-1)$ is defined, then it must be the case that $\mathcal{K}'(k, |\mathcal{K}'(k)|-1) = (\_, t, \emptyset)$, hence it must be the case that $i < |\mathcal{K}'(k)| - 1$. Because $t \notin \mathcal{K}$, then by the definition of update it must be the case that $(\mathtt{r}, k, \_) \in \mathcal{F}$, $\mathcal{K}(k,i) = (\_, t', \_)$ and $i = \max(u(k))$; this also implies that $t' = \max_{\mathsf{WW}(k)} \{\mathsf{w}(\mathcal{K}(k,i)) \mid i \in u(k)\}$.
- If $t'' \in T_{\mathcal{G}}$, then it must be the case that $t'' \neq t$. In this case, it also must exist an index $i = 0, \cdots, |\mathcal{K}'(k)| - 1$ such that $\mathcal{K}'(k,i) = (\_, t', \{t''\} \cup \_)$. As in the previous case, we note that $i < |\mathcal{K}'(k)| - 1$, which together with the fact that $t'' \neq t$ leads to $\mathcal{K}(k,i) = (\_, t', \{t''\} \cup \_)$. It follows that $t' \xrightarrow{\mathsf{WR}_{\mathcal{G}}(k)} t''$.

(3) Similar to $\mathsf{WR}(k)_{\mathcal{G}}$, there are two cases that either the t'' already exists in the dependency graph before, or it is the newly committed transaction.

– Suppose that $t' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(k)} t''$. By definition, there exist two indexes i, j such that $\mathcal{K}(k,i) = (\_, t', \_)$, $\mathcal{K}(k,j) = (\_, t'', \_)$ and $i < j$. The definition of update ensures that $\mathcal{K}'(k,i) = (\_, t', \_)$, $\mathcal{K}'(k,j) = (\_, t'', \_)$, and because $i < j$ we obtain that $t' \xrightarrow{\mathsf{WW}_{\mathcal{G}'}(k)} t''$.

– Suppose that $(\mathtt{w}, k, \_) \in \mathcal{F}$. Then $\mathcal{K}'(k, |\mathcal{K}(k)|) = (\_, t, \_)$. Let $t' \in T_{\mathcal{G}}$; by definition there exists an index $i = 0, \cdots, |\mathcal{K}(k)|$ such that $\mathcal{K}(k,i) = (\_, t', \_)$. It follows that $\mathcal{K}'(k,i) = (\_, t', \_)$, and because $i < |\mathcal{K}(k)|$, then we have that $t' \xrightarrow{\mathsf{WW}_{\mathcal{G}'}(k)} t$.

Conversely, suppose that $t' \xrightarrow{\mathsf{WW}_{\mathcal{G}'}(k)} t''$. Because $T_{\mathcal{G}'} = T_{\mathcal{G}} \cup \{t\}$, we have two possibilities. Either $t'' = t$, or $t'' \in T_{\mathcal{G}}$.

- If $t'' = t$, then it must be the case that $(\mathtt{w}, k, \_) \in_{\mathcal{G}'} t$, or equivalently there exists an index $i = 0, \cdots, |\mathcal{K}'(k)| - 1$ such that $\mathcal{K}'(k,i) = (\_, t, \_)$. Because $t \notin \mathcal{K}$, and because for any $i = 0, \cdots, |\mathcal{K}(k)| - 1$, $\mathcal{K}'(k,i) = (\_, t, \_) \Rightarrow \mathcal{K}(k,i) = (\_, t, \_)$, then it necessarily has to be $i = \mathcal{K}'(k)| - 1$. According to the definition of update, this is possible only if $(\mathtt{w}, k, \_) \in \mathcal{F}$. Finally, note that because $t' \xrightarrow{\mathsf{WW}_{\mathcal{G}'}(k)} t$, then there exists an index $j < |\mathcal{K}'(k,i)| - 1$ such that $\mathcal{K}'(k,j) = (\_, t', \_)$. The fact that $j < |\mathcal{K}'(k,i)| - 1$ we obtain that $\mathcal{K}(k,j) = (\_, t', \_)$, or equivalently $t' = \mathsf{w}(\mathcal{K}(k, \_))$.
- If $t'' \in T_{\mathcal{G}}$, then there exist two indexes i, j such that $j < |\mathcal{K}'(k,j)|-1$, $\mathcal{K}'(k,j) = (\_, t'', \_)$, $i < j$, and $\mathcal{K}'(k,i) = (\_, t', \_)$. It is immediate to observe that $\mathcal{K}(k,i) = (\_, t', \_)$, $\mathcal{K}(k,j) = (\_, t'', \_)$, from which $t' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(k)} t''$ follows.

D.3 Abstract Execution Traces to KV-Store Traces

We show to construct, given an abstract execution $\mathcal{X}$, a set of $\mathtt{ET}_\top$-traces $\mathsf{KVtrace}(\mathtt{ET}_\top, \mathcal{X})$ in normal form such that for any $\tau \in \mathsf{KVtrace}(\mathtt{ET}_\top, \mathcal{X})$, the trace τ satisfies $\mathsf{lastConf}(\tau) = (\mathcal{K}_\mathcal{X}, \_)$. We first define the $\mathsf{cut}(\mathcal{X}, n)$ function in Def. 27 which gives the prefix of the first n transactions of the abstract execution $\mathcal{X}$. The $\mathsf{cut}(\mathcal{X}, n)$ function is very useful for later discussion.

Definition 27. *Let $\mathcal{X}$ be an abstract execution, let $n = |T_\mathcal{X}|$, and let $\{t_i\}_{i=1}^n \subseteq T_\mathcal{X}$ be such that $t_i \xrightarrow{\mathsf{AR}_\mathcal{X}} t_{i+1}$. The* cut *of the first n transactions from an abstract execution $\mathcal{X}$ is defined as the follows:*

$$\begin{aligned}\mathsf{cut}(\mathcal{X}, 0) &\triangleq ([], \emptyset, \emptyset)\\ \mathsf{cut}(\mathcal{X}, i+1) &\triangleq \mathsf{extend}\big(\mathsf{cut}(\mathcal{X}, i), t_{i+1}, \mathsf{VIS}_\mathcal{X}^{-1}(t_{i+1}), \mathscr{T}_\mathcal{X}(t_{i+1})\big)\end{aligned}$$

Proposition 12 (Well-defined cut). *For any abstract execution $\mathcal{X}$, $\mathcal{X} = \mathsf{cut}(\mathcal{X}, |T_\mathcal{X}|)$.*

Proof. This is an instantiation of Lemma 3 by choosing $i = |T_\mathcal{X}|$.

Lemma 3 (Prefix). *For any abstract execution $\mathcal{X}$, and index $i : i \leq j \leq |T_\mathcal{X}|$, if $T_\mathcal{X} = \{t_i\}_{i=1}^n$ be such that $t_i \xrightarrow{\mathsf{AR}_\mathcal{X}} t_{i+1}$, then $\mathsf{cut}(\mathcal{X}, i) = \mathcal{X}_i$ where*

$$\begin{aligned}\mathscr{T}_{\mathcal{X}_i}(t) &= \begin{cases}\mathscr{T}_\mathcal{X}(t) & \text{if } \exists j \leq i.\ t = t_j\\ \textit{undefined} & \textit{otherwise}\end{cases}\\ \mathsf{VIS}_{\mathcal{X}_i} &= \left\{(t, t') \in T_{\mathcal{X}_i} \;\middle|\; t \xrightarrow{\mathsf{VIS}_\mathcal{X}} t'\right\}\\ \mathsf{AR}_{\mathcal{X}_i} &= \left\{(t, t') \in T_{\mathcal{X}_i} \;\middle|\; t \xrightarrow{\mathsf{AR}_\mathcal{X}} t'\right\}\end{aligned}$$

Proof. Fix an abstract execution $\mathcal{X}$. We prove by induction on $i = |T_\mathcal{X}|$.

- Base case: $i = 0$. Then $\mathscr{T}_{\mathcal{X}'} = [], \mathsf{VIS}_{\mathcal{X}'} = \emptyset$, $\mathsf{AR}_{\mathcal{X}'} = \emptyset$, which leads to $\mathcal{X}' = \mathsf{cut}(\mathcal{X}, 0)$.
- Inductive case: $i = i' + 1$. Assume that $\mathsf{cut}(\mathcal{X}, i') = \mathcal{X}_{i'}$. We prove the following:
  - $\mathscr{T}_{\mathsf{cut}(\mathcal{X},i)} = \mathscr{T}_{\mathcal{X}_i}$. By definition,
  $$\mathscr{T}_{\mathsf{cut}(\mathcal{X},i)} = \mathscr{T}_{\mathsf{cut}(\mathcal{X},i')}[t_i \mapsto \mathscr{T}_\mathcal{X}(t_i)]\mathscr{T}_{\mathcal{X}_{i'}}[t_i \mapsto \mathscr{T}_\mathcal{X}](t_i) = \mathscr{T}_{\mathcal{X}_i}$$
  - $\mathsf{VIS}_{\mathsf{cut}(\mathcal{X},i)} = \mathsf{VIS}_{\mathcal{X}_i}$. Note that, by inductive hypothesis, $T_{\mathsf{cut}(\mathcal{X},i')} = T_{\mathcal{X}_{i'}} = \{t_j\}_{j=1}^{i'}$. We have that
  $$\begin{aligned}\mathsf{VIS}_{\mathsf{cut}(\mathcal{X},i)} &= \mathsf{VIS}_{\mathsf{cut}(\mathcal{X},i')} \cup \{(t_j, t_i) \in \mathsf{VIS}_\mathcal{X} \mid 1 \leq j \leq i'\}\\ &= \mathsf{VIS}_{\mathcal{X}_{i'}} \cup \{(t_j, t_i) \in \mathsf{VIS}_\mathcal{X} \mid 1 \leq j \leq i'\}\\ &= \{(t_{j'}, t_j) \in \mathsf{VIS}_\mathcal{X} \mid 1 \leq j \leq i'\} \cup \{(t_j, t_i) \in \mathsf{VIS}_\mathcal{X} \mid 1 \leq j \leq i'\}\\ &= \{(t_{j'}, t_j) \in \mathsf{VIS}_\mathcal{X} \mid 1 \leq j \leq i'\}\\ &= \mathsf{VIS}_{\mathcal{X}_i}\end{aligned}$$

- $\mathsf{AR}_{\mathsf{cut}(\mathcal{X},i)} = \mathsf{AR}_{\mathcal{X}_i}$. It follows the same way as the above.

Let $\textsc{ClientID}(\mathcal{X}) \triangleq \{cl \mid \exists n.\ t^n_{cl} \in T_{\mathcal{X}}\}$. Given an abstract execution $\mathcal{X}$, a client cl and an index $i : 0 \leq i < |T_{\mathcal{X}}|$, the function $\mathsf{nextTid}(\mathcal{X}, cl, i) \triangleq \min_{\mathsf{AR}_{\mathcal{X}}} \left\{ t^j_{cl} \;\middle|\; t^n_{cl} \notin T_{\mathsf{cut}(\mathcal{X},i)} \right\}$. Note that $\mathsf{nextTid}(\mathcal{X}, cl, i)$ could be undefined. The conversion from abstract execution tests to $\mathtt{ET}$ traces is in Def. 28.

Definition 28. *Given an abstract execution $\mathcal{X}$ and an index $i : 0 \leq i < |T_{\mathcal{X}}|$, the function $\mathsf{KVtrace}(\mathtt{ET}_\top, \mathcal{X}, i)$ is defined as the smallest set such that:*

- $(\mathcal{K}_0, \lambda cl \in \textsc{ClientID}(\mathcal{X}).\lambda k.\, \{0\}) \in \mathsf{KVtrace}(\mathtt{ET}_\top, \mathcal{X}, 0)$,
- *suppose that $\tau \in \mathsf{KVtrace}(\mathtt{ET}_\top, \mathcal{X}, i)$ for some i. Let*
  - $t = \min_{\mathsf{AR}_{\mathcal{X}}}(T_{\mathcal{X}} \setminus T_{\mathsf{cut}(\mathcal{X},i)})$,
  - cl, n *be such that* $t = t^n_{cl}$,
  - $u = \mathsf{getView}\big(\mathcal{X}, \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl})\big)$,
  - $u' = \mathsf{getView}(\mathcal{X}, T)$, *where T is an arbitrary subset of $T_{\mathcal{X}}$ if* $\mathsf{nextTid}(\mathcal{X}, cl, i+1)$ *is undefined, or is such that* $T \subseteq (\mathsf{AR}^{-1}_{\mathcal{X}})^?(t) \cap \mathsf{VIS}^{-1}_{\mathcal{X}}(\mathsf{nextTid}(cl, i+1))$,
  - $\mathcal{F} = \mathscr{T}_{\mathcal{X}}(t)$,
  - $(\mathcal{K}_\tau, \mathcal{U}_\tau) = \mathsf{lastConf}(\tau)$, *and*
  - $\mathcal{K} = \mathsf{update}(\mathcal{K}_\tau, u, \mathcal{F}, t)$.

  Then

$$\Big(\tau \xrightarrow{(cl,\varepsilon)}_{\mathtt{ET}_\top} (\mathcal{K}_\tau, \mathcal{U}_\tau[cl \mapsto u]) \xrightarrow{(cl,\mathcal{F})}_{\mathtt{ET}_\top} (\mathcal{K}, \mathcal{U}_\tau[cl \mapsto u'])\Big) \in \mathsf{KVtrace}(\mathtt{ET}_\top, \mathcal{X}, i{+}1)$$

Last, the function $\mathsf{KVtrace}(\mathtt{ET}_\top, \mathcal{X}) \triangleq \mathsf{KVtrace}(\mathtt{ET}_\top, \mathcal{X}, |T_{\mathcal{X}}|)$.

Proposition 13 (Abstract executions to trace $\mathtt{ET}_\top$). *Given an abstract execution $\mathcal{X}$ satisfying $\mathsf{RP}_{\mathsf{LWW}}$, and a trace $\tau \in \mathsf{KVtrace}(\mathtt{ET}_\top, \mathcal{X})$, then* $\mathsf{lastConf}(\tau) = (\mathcal{K}_{\mathcal{X}}, \_)$ *and* $\mathcal{K}_{\mathcal{X}} \in \mathsf{CM}(\mathtt{ET}_\top)$.

Proof. Let $\mathcal{X}$ be an abstract execution that satisfies the last write wins policy. Let $n = |T_{\mathcal{X}}|$. Fix $i = 0, \cdots, n$, and let $\tau \in \mathsf{KVtrace}(\mathtt{ET}_\top, \mathcal{X}, i)$. We prove, by induction on i, that $\tau \in \mathsf{CM}(\mathtt{ET}_\top)$, and $\mathsf{lastConf}(\tau) = (\mathcal{K}_{(\mathsf{cut}(\mathcal{X},i)}, \_)$. Then the result follows from Prop. 12.

- Base case: $i = 0$. By definition, $\tau = (\mathcal{K}_0, \mathcal{U}_0)$, where $\mathcal{U}_0 = \lambda cl \in \textsc{ClientID}(\mathcal{X}).\lambda k.\, \{0\}$. Clearly, we have that $\tau \in \mathsf{CM}(\mathtt{ET}_\top)$.
- Inductive case: $i = i' + 1$. Let $t_i = \min_{\mathsf{AR}_{\mathcal{X}}}(T_{\mathcal{X}} \setminus T_{\mathsf{cut}(\mathcal{X},i')})$, and suppose that $t_i = t^m_{cl}$ for some client cl and index m. Fix $u = \mathsf{getView}\big(\mathcal{X}, \mathsf{VIS}^{-1}_{\mathcal{X}}(t_i)\big)$, and $\mathcal{F} = \mathscr{T}_{\mathcal{X}}(t_i)$. We prove that there exists a trace $\tau' \in \mathsf{KVtrace}(\mathtt{ET}_\top, \mathcal{X}, i')$ and a set T such that:

  (1) if $\mathsf{nextTid}(cl, \mathcal{X}, i)$ is undefined then $T \subseteq T_{\mathcal{X}}$, otherwise

$$T \subseteq \mathsf{VIS}^{-1}_{\mathcal{X}}(\mathsf{nextTid}(cl, \mathcal{X}, i)) \cap (\mathsf{AR}^{-1}_{\mathcal{X}})^?(t_i)$$

(2) the new trace τ such that

$$\tau = \tau' \xrightarrow{(cl,\varepsilon)} (\mathcal{K}_{\tau'}, \mathcal{U}_{\tau'}[cl \mapsto u]) \xrightarrow{(cl,\mathcal{F})} (\mathcal{K}, \mathcal{U}_{\tau'}[cl \mapsto u'])$$

where $(\mathcal{K}_{\tau'}, \mathcal{U}_{\tau'}) = \mathsf{lastConf}(\tau')$, and $\mathcal{K} = \mathsf{update}(\mathcal{K}_{\tau'}, u, \mathcal{F}, t_i)$, and $u' = \mathsf{getView}(\mathcal{X}, T)$.

By inductive hypothesis, we may assume that $\tau' \in \mathsf{CM}(\mathtt{ET}_\top)$, and $\mathcal{K}_{\tau'} = \mathcal{K}_{\mathsf{cut}(\mathcal{X},i')}$. We prove the following facts:

(1) $\mathcal{K} = \mathcal{K}_{\mathsf{cut}(\mathcal{X},i)}$. Because of Prop. 11 and Prop. 12, we obtain

$$\begin{aligned}
\mathcal{K} &= \mathsf{update}(\mathcal{K}_{\tau'}, u, \mathcal{F}, t_i) \\
&= \mathsf{update}\big(\mathcal{K}_{\mathsf{cut}(\mathcal{X},i')}, \mathsf{getView}\big(\mathcal{X}, \mathsf{VIS}_{\mathcal{X}}^{-1}(t_i)\big), \mathscr{T}_{\mathcal{X}}(t_i), t_i\big) \\
&= \mathcal{K}_{\mathsf{extend}\left(\mathsf{cut}(\mathcal{X},i'), \mathsf{VIS}_{\mathcal{X}}^{-1}(t_i), t_i, \mathscr{T}_{\mathcal{X}}(t_i)\right)} \\
&= \mathcal{K}_{\mathsf{extend}(\mathsf{cut}(\mathcal{X},i))}
\end{aligned}$$

(2) $(\mathcal{K}_{\tau'}, \mathcal{U}_{\tau'}) \xrightarrow{(cl,\varepsilon)} (\mathcal{K}_{\tau'}, \mathcal{U}_{\tau'}[cl \mapsto u])$. It suffices to prove that $\mathcal{U}_{\tau'}(cl) \sqsubseteq u$ for any key k. By Lemma 3 we have that $T_{\mathsf{cut}(\mathcal{X},i')} = \{t_j\}_{j=1}^{i'}$, for some $t_1, \cdots, t_{i'}$ such that whenever $1 \leq j < j' \leq i'$, then $t_j \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t_{j'}$. We consider two possible cases:

- For all $j : 1 \leq j \leq i'$, and $h \in \mathbb{N}$, then $t_j \neq t_{cl}^h$. In this case we have that no transition contained in τ' has the form $(\_, \_) \xrightarrow{(cl,\_)} (\_, \_)$, from which it is possible to infer that $\mathcal{U}_{\tau'}(cl) = \lambda k.\{0\}$. Because $u = \mathsf{getView}\big(\mathcal{X}, \mathsf{VIS}_{\mathcal{X}}^{-1}(t_i)\big)$, then by definition we have that $0 \in u\,(k)$ for all keys $k \in \textsc{Keys}$. It follows that $\mathcal{U}_{\tau'}(cl) \sqsubseteq u$.
- There exists an index $j : 1 \leq j \leq i'$ and an integer $h \in \mathbb{N}$ such that $t_j = t_{cl}^h$. Without loss of generality, let j be the largest such index. It follows that the last transition in τ' of the form $(\_, \_) \xrightarrow{(cl,\mathcal{F}_j)} (\_, \mathcal{U}_{\mathsf{pre}})$ is such that $\mathcal{U}_{\mathsf{pre}}(cl) = \mathsf{getView}(\mathcal{X}, T_{\mathsf{pre}})$, for some $T_{\mathsf{pre}} \subseteq \mathsf{VIS}_{\mathcal{X}}^{-1}(t_i) \cap (\mathsf{AR}_{\mathcal{X}}^{-1})^?(t_j)$. This is because $\mathsf{nextTid}(cl, \mathcal{X}, j)$ is defined and equal to t_i. Furthermore, because the trace τ' is in normal form by construction, in τ' a transition of the form $(\_, \_) \xrightarrow{(cl,\varepsilon)}_{\mathtt{ET}_\top} (\_, \_)$ is always followed by a transition of the form $(\_, \_) \xrightarrow{(cl,\mathcal{F}')}_{\mathtt{ET}_\top} (\_, \_)$. Because we assume that the last transition where client cl executes a transaction in τ' has the form $(\_, \_) \xrightarrow{(cl,\mathcal{F}_j)}_{\mathtt{ET}_\top} (\_, \mathcal{U}_{\mathsf{pre}})$, then the latter is also the last transition for client cl in τ' (i.e. including both execution of transactions and view updates). It follows that $\mathcal{U}_{\tau'}(cl) = \mathcal{U}_{\mathsf{pre}}(cl)$, and in particular $\mathcal{U}_{\tau'}(cl) = \mathsf{getView}(\mathcal{X}, T_{\mathsf{pre}})$. By definition, $T_{\mathsf{pre}} \subseteq \mathsf{VIS}_{\mathcal{X}}^{-1}(t_i) \cap (\mathsf{AR}_{\mathcal{X}}^{-1})^?(t_j) \subseteq \mathsf{VIS}_{\mathcal{X}}^{-1}(t_i)$. By Lemma 4, we have that $\mathcal{U}_{\tau'}(cl) = \mathsf{getView}(\mathcal{X}, T_{\mathsf{pre}}) \sqsubseteq \mathsf{getView}\big(\mathcal{X}, \mathsf{VIS}_{\mathcal{X}}^{-1}(t_i)\big) = u$, as we wanted to prove.

(3) $(\mathcal{K}_{\tau'}, \mathcal{U}_{\tau'}[cl \mapsto u]) \xrightarrow{(cl,\mathcal{F})}_{\mathtt{ET}_\top} (\mathcal{K}, \mathcal{U}_{\tau'}[cl \mapsto u'])$. It suffices to show that $\mathtt{ET}_\top \vdash (\mathcal{K}_{\tau'}, u) \triangleright \mathcal{F} : (\mathcal{K}, u')$. That is, it suffices to show that $u \in$

$\textsc{Views}(\mathcal{K}_{\tau'})$, $u' \in \textsc{Views}(\mathcal{K})$, and whenever $(\mathtt{r}, k, v) \in \mathcal{F}$, then $\max_<(u(k)) = (v, \_, \_)$. The first two facts are a consequence of Lemma 5, $\mathcal{K}_{\tau'} = \mathcal{K}_{\mathsf{cut}(\mathcal{X},i')}$, and $\mathcal{K}_{\mathsf{cut}(\mathcal{X},i)}$. The last one that if $(\mathtt{r}, k, v) \in \mathcal{F}$ then $\max_<(u(k)) = (v, \_, \_)$ follows the fact that $\mathcal{X}$ satisfies the last write wins policy and the fact that $u = \mathsf{getView}\big(\mathsf{VIS}^{-1}_{\mathcal{X}}(t_i)\big)$.

Lemma 4 (Monotonic getView). *Let $\mathcal{X}$ be an abstract execution, and let $T_1 \subseteq T_2 \subseteq T_{\mathcal{X}}$. Then $\mathsf{getView}(\mathcal{X}, T_1) \sqsubseteq \mathsf{getView}(\mathcal{X}, T_2)$.*

Proof. Fix $k \in \textsc{Keys}$. By definition

$$\begin{aligned}\mathsf{getView}(\mathcal{X}, T_1)(k) &= \{0\} \cup \{i \mid \mathsf{w}(\mathcal{K}_{\mathcal{X}}(k,i)) \in T_1\}\\ &\subseteq \{0\} \cup \{i \mid \mathsf{w}(\mathcal{K}_{\mathcal{X}}(k,i)) \in T_2\}\\ &= \mathsf{getView}(\mathcal{X}, T_2)(k)\end{aligned}$$

then it follows that $\mathsf{getView}(\mathcal{X}, T_1) \sqsubseteq \mathsf{getView}(\mathcal{X}, T_2)$.

Lemma 5 (Valid view on cut of abstract execution). *Let $\mathcal{X}$ be an abstract execution, with $T_{\mathcal{X}} = \{t_i\}_{i=1}^n$ for $n = |T_{\mathcal{X}}|$, and $i : 0 \leq i < n$ such that $t_i \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t_{i+1}$. Assuming $T_0 = \emptyset$, and $T_i \subseteq (\mathsf{AR}^{-1})^?(t_i)$ for $i : 0 \leq i \leq n$, then $\mathsf{getView}(\mathcal{X}, T_i) \in \textsc{Views}\big(\mathcal{K}_{\mathsf{cut}(\mathcal{X},i)}\big)$.*

Proof. We prove by induction on the index i.

- Base case: $i = 0$. It follows $T_0 = \emptyset$, and $\mathsf{getView}(\mathcal{X}, T_0) = \lambda k.\{0\}$. We also have that $\mathcal{K}_{\mathsf{cut}(\mathcal{X},0)} = \lambda k.(v_0, t_0, \emptyset)$, hence it is immediate to see that $\mathsf{getView}(\mathcal{X}, T_0) \in \textsc{Views}\big(\mathcal{K}_{\mathsf{cut}(\mathcal{X},0)}\big)$.
- Inductive case: $i = i' + 1$. Suppose that for any $T \subseteq (\mathsf{AR}^{-1}_{\mathcal{X}})^?(t_{i'})$, then $\mathsf{getView}(\mathcal{X}, T) \in \textsc{Views}\big(\mathcal{K}_{\mathsf{cut}(\mathcal{X},i)}\big)$. Let consider the set T_i. Note that, because of Prop. 11, we have that

  $$\mathcal{K}_{\mathsf{cut}(\mathcal{X},i)} = \mathcal{K}_{\mathsf{extend}\left(\mathsf{cut}(\mathcal{X},i'),t_i,\mathsf{VIS}^{-1}_{\mathcal{X}}(t_i),\mathscr{T}_{\mathcal{X}}(t_i)\right)} = \mathsf{update}\big(\mathcal{K}_{\mathsf{cut}(\mathcal{X},i')}, \mathsf{getView}\big(\mathsf{VIS}^{-1}_{\mathcal{X}}(t_i)\big), \mathscr{T}_{\mathcal{X}}(t_i), t_i\big)$$

  There are two possibilities:
  - $t_i \notin T_i$, where case $T_i \subseteq (\mathsf{AR}^{-1}_{\mathcal{X}})^?(t_{i'})$. From the inductive hypothesis we get $\mathsf{getView}(\mathcal{X}, T_i) \in \textsc{Views}\big(\mathcal{K}_{\mathsf{cut}(\mathcal{X},i')}\big)$. Note that $\mathcal{K}_{\mathsf{cut}(\mathcal{X},i')}$ only contains the transactions identifiers from t_1 to $t_{i'}$; in particular, it does not contain t_i. Because $\mathcal{K}_{\mathsf{cut}(\mathcal{X},i)} = \mathsf{update}\big(\mathcal{K}_{\mathsf{cut}(\mathcal{X},i')}, \_, \_, t_i\big)$, then by Lemma 6 we have that $\mathsf{getView}(\mathcal{X}, t_i) \in \textsc{Views}\big(\mathcal{K}_{\mathsf{cut}(\mathcal{X},i)}\big)$.
  - $t \in T_i$. Note that for any key k such that $(\mathtt{w}, k, \_) \notin \mathscr{T}_{\mathcal{X}}(t_i)$, then $\mathsf{getView}(\mathcal{X}, T_i)(k) = \mathsf{getView}(\mathcal{X}, T_i \setminus \{t_i\})(k)$; and for any key k such that $(\mathtt{w}, k, \_) \in \mathscr{T}_{\mathcal{X}}(t_i)$, then $\mathsf{getView}(\mathcal{X}, T_i)(k) = \mathsf{getView}(\mathcal{X}, T_i \setminus \{t_i\})(k) \cup \{j \mid \mathsf{w}(\mathcal{K}_{\mathcal{X}}(k,i)) = t_j\}$. In the last case, the index j must be such that $j < |\mathcal{K}_{\mathsf{cut}(\mathcal{X},i)}| - 1$, because we know that $t_i \in \mathcal{K}_{\mathsf{cut}(\mathcal{X},i)}$. It follows from this fact and the inductive hypothesis, that $\mathsf{getView}(\mathcal{X}, T_i) \in \textsc{Views}\big(\mathcal{K}_{\mathsf{cut}(\mathcal{X},i)}\big)$.

Lemma 6 (update preserving views). *Given a kv-store $\mathcal{K}$, a transactions $t \notin \mathcal{K}$, views $u, u' \in \textsc{Views}(\mathcal{K})$, and set of operations $\mathcal{F}$, then $u \in \mathsf{update}(\mathcal{K}, u', \mathcal{F}, t)$.*

Proof. Immediate from the definition of update. Note that $t \notin \mathcal{K}$ ensures that u still satisfies (atomic) with respect to the new kv-store $\mathsf{update}(\mathcal{K}, u', \mathcal{F}, t)$.

E The Sound and Complete Constructors of the KV-Store Semantics with Respect to Abstract Executions

In this Section we first define the set of ET-traces generated by a program P. Then we prove correctness our semantics on kv-stores, meaning that if a program P executing under the execution test ET terminates in a state $(\mathcal{K}, \_)$, then $\mathcal{K} \in \mathsf{CM}(\mathtt{ET})$.

E.1 Traces of Programs under KV-Stores

The $\mathsf{Ptraces}(\mathtt{ET}, \mathtt{P})$ is the set of all possible traces generated by the program P starting from the initial configuration $(\mathcal{K}_0, \mathcal{U}_0)$.

Definition 29. *Given an execution test* ET *a program* P *and a state* $(\mathcal{K}, \mathcal{U}, \mathcal{E})$*, the* $\mathsf{Ptraces}(\mathtt{ET}, \mathtt{P}, \mathcal{K}, \mathcal{U}, \mathcal{E})$ *function is defined as the smallest set such that:*

- $(\mathcal{K}, \mathcal{U}) \in \mathsf{Ptraces}(\mathtt{ET}, \mathtt{P}, \mathcal{K}, \mathcal{U}, \mathcal{E})$
- *if* $\tau \in \mathsf{Ptraces}(\mathtt{ET}, \mathtt{P}', \mathcal{K}', \mathcal{U}', \mathcal{E}')$ *and* $((\mathcal{K}, \mathcal{U}, \mathcal{E}), \mathtt{P}) \xrightarrow{(cl,\iota)}_{\mathtt{ET}} (\mathcal{K}', \mathcal{U}', \mathcal{E}')$*, then*

$$\tau \in \mathsf{Ptraces}(\mathtt{ET}, \mathtt{P}, \mathcal{K}, \mathcal{U}, \mathcal{E}')$$

- *if* $\tau \in \mathsf{Ptraces}(\mathtt{ET}, \mathtt{P}', \mathcal{K}', \mathcal{U}', \mathcal{E}')$ *and* $(\mathcal{K}, \mathcal{U}, \mathcal{E}), \mathtt{P}) \xrightarrow{(cl,u,\mathcal{F})} ((\mathcal{K}', \mathcal{U}', \mathcal{E}'), \mathtt{P}')$*, then*

$$\Big((\mathcal{K}, \mathcal{U}) \xrightarrow{(cl,\varepsilon)}_{\mathtt{ET}} (\mathcal{K}, \mathcal{U}[cl \mapsto u]) \xrightarrow{(cl,\mathcal{F})}_{\mathtt{ET}} \tau \Big) \in \mathsf{Ptraces}(\mathtt{ET}, \mathtt{P}, \mathcal{K}, \mathcal{U}, \mathcal{E})$$

The set of traces generated by a program P *under the execution test* ET *is then defined as* $\mathsf{Ptraces}(\mathtt{ET}, \mathtt{P}) \triangleq \mathsf{Ptraces}(\mathtt{ET}, \mathtt{P}, \mathcal{K}_0, \mathcal{U}_0, \mathcal{E}_0)$*, where* $\mathcal{U}_0 = \lambda cl \in \mathrm{dom}(\mathtt{P}).\lambda k.\, \{0\}$*, and* $\mathcal{E}_0 = \lambda cl \in \mathrm{dom}(\mathtt{P}).\lambda a.0$*.*

Proposition 14. *For any program* P *and execution test* ET*,* $\mathsf{Ptraces}(\mathtt{ET}, \mathtt{P}) \subseteq \mathsf{conf}(\mathtt{ET})$ *and* $\tau \in \mathsf{Ptraces}(\mathtt{ET}, \mathtt{P})$ *is in normal form.*

Proof. First, by the definition of Ptraces, it only constructs trace in normal form. It is easy to prove that for any trace τ in $\mathsf{Ptraces}(\mathtt{ET}, \mathtt{P})$, by induction on the trace length, the trace is also in $\mathsf{conf}(\mathtt{ET})$.

Corollary 1. *If a trace in the following form*

$$(\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}_0), \mathtt{P}) \rightarrow_{\mathtt{ET}} \cdots \rightarrow_{\mathtt{ET}} (\mathcal{K}, \mathcal{U}, \mathcal{E}, \lambda cl \in \mathrm{dom}(\mathtt{P}).\mathtt{skip})$$

then $\mathcal{K} \in \mathsf{CM}(\mathtt{ET})$*.*

Proof. By the definition of Ptraces, there exists a corresponding trace $\tau \in \mathsf{Ptraces}(\mathtt{ET}, \mathtt{P})$. By Prop. 14, such trace $\tau \in \mathsf{conf}(\mathtt{ET})$, therefore $\mathcal{K} \in \mathsf{CM}(\mathtt{ET})$ by definition of $\mathsf{CM}(\mathtt{ET})$.

Similar to $[\![\mathtt{P}]\!]_{(\mathsf{RP},\mathcal{A})}$, the function $[\![\mathtt{P}]\!]_{\mathtt{ET}}$ is defined as the following:

$$[\![\mathtt{P}]\!]_{\mathtt{ET}} = \left\{\mathcal{K} \mid (\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}_0), \mathtt{P} \xrightarrow{}^*_{\mathtt{ET}} (\mathcal{K}, \_, \_), \mathtt{P}_f)\right\}$$

where $\mathcal{E}_0 = \lambda cl \in \mathrm{dom}(\mathtt{P}).\lambda \mathtt{x}.0$ and $\mathtt{P}_f = \lambda cl \in \mathrm{dom}(\mathtt{P}).\mathtt{skip}$.

Proposition 15. *For any program* P *and execution test* ET*:* $[\![\mathtt{P}]\!]_{\mathtt{ET}} = [\![\mathtt{P}]\!]_{\mathtt{ET}_\top} \cap \mathsf{CM}(\mathtt{ET})$.

Proof. We prove a stronger result that for any program P and execution test ET, $\mathsf{Ptraces}(\mathtt{ET}, \mathtt{P}) = \mathsf{Ptraces}(\mathtt{ET}_\top, \mathtt{P}) \cap \mathsf{conf}(\mathtt{ET})$. It is easy to see $\mathsf{Ptraces}(\mathtt{ET}, \mathtt{P}) \subseteq \mathsf{Ptraces}(\mathtt{ET}_\top, \mathtt{P})$. By Prop. 14, we know $\mathsf{Ptraces}(\mathtt{ET}, \mathtt{P}) \subseteq \mathsf{conf}(\mathtt{ET})$. Therefore $\mathsf{Ptraces}(\mathtt{ET}, \mathtt{P}) \subseteq \mathsf{Ptraces}(\mathtt{ET}_\top, \mathtt{P}) \cap \mathsf{conf}(\mathtt{ET})$.

Let consider a trace τ in $\mathsf{Ptraces}(\mathtt{ET}_\top, \mathtt{P}) \cap \mathsf{conf}(\mathtt{ET})$. By inductions on the length of trace, every step that commits a new transaction must satisfy ET as $\tau \in \mathsf{conf}(\mathtt{ET})$. It also reduce the program P since $\tau \in \mathsf{Ptraces}(\mathtt{ET}_\top, \mathtt{P})$. By the definition $\mathsf{Ptraces}(\mathtt{ET}, \mathtt{P})$, we can construct the same trace τ so that $\tau \in \mathsf{Ptraces}(\mathtt{ET}, \mathtt{P})$.

E.2 Adequate of KV-Store Semantic

Our main aim is to prove that for any program P, the set of kv-stores generated by P under ET corresponds to all the possible abstract executions that can be obtained by running P on a database that satisfies the axiomatic definition $\mathcal{A}$. In this sense, we aim to establish that our operational semantics is *adequate*.

More precisely, suppose that a given execution test ET captures precisely a consistency model defined in the axiomatic style, using a set of axioms $\mathcal{A}$ and a resolution policy RP over abstract executions. That is, for any abstract execution $\mathcal{X}$ that satisfies the axioms $\mathcal{A}$ and the resolution policy RP, then $\mathsf{KVtrace}(\mathtt{ET}_\top, \mathcal{X}) \cap \mathsf{CM}(\mathtt{ET}) \neq \emptyset$; and for any $\tau \in \mathsf{CM}(\mathtt{ET})$, there exists an abstract execution $\mathcal{X} \in \mathsf{absExec}(\tau)$ that satisfies the axioms $\mathcal{A}$ and the resolution policy RP.

We now consider the program P. The Prop. 16 and Prop. 17 show the connection between reduction steps between the last write win resolution policy $(\mathsf{RP}_{\mathsf{LWW}}, \emptyset)$ and the most permissive execution test $\mathtt{ET}_\top$.

Proposition 16 (Permissive execution test to last write win). *Suppose that*

$$(\mathcal{K}, \mathcal{U}, \mathcal{E}), \mathtt{P} \xrightarrow{(cl,u,\mathcal{F})}_{\mathtt{ET}_\top} (\mathcal{K}', \mathcal{U}', \mathcal{E}'), \mathtt{P}'$$

Assuming an abstract execution $\mathcal{X}$ such that $\mathcal{K}_\mathcal{X} = \mathcal{K}$, and a set of read-only transactions $T \subseteq T_\mathcal{X}$, then there exists an abstract execution $\mathcal{X}'$ such that $\mathcal{K}_{\mathcal{X}'} = \mathcal{K}'$, and

$$(\mathcal{X}, \mathcal{E}), \mathtt{P} \xrightarrow{(cl, T \cup \mathsf{visTx}(\mathcal{K},u), \mathcal{F})}_{(\mathsf{RP}_{\mathsf{LWW}}, \emptyset)} (\mathcal{X}', \mathcal{E}'), \mathtt{P}'$$

Proof. Suppose that $(\mathcal{K}, \mathcal{U}, \mathcal{E}), \mathtt{P} \xrightarrow{(cl,u,\mathcal{F})}_{\mathtt{ET}_\top} (\mathcal{K}', \mathcal{U}', \mathcal{E}), \mathtt{P}'$. This transition can only be inferred by applying Rule PSingleThread, meaning that

- $\mathtt{P}\,(cl) \mapsto \mathtt{C}$ for some command $\mathtt{C}$,
- $cl \vdash (\mathcal{K}, \mathcal{U}\,(cl)\,, \mathcal{E}(cl)), \mathtt{C} \xrightarrow{(cl,u,\mathcal{F})}_{\mathtt{ET}_\top} (\mathcal{K}', u', s'), \mathtt{C}'$ for some u', s', and
- $\mathcal{U}' = \mathcal{U}[cl \mapsto u']$, $\mathcal{E}' = \mathcal{E}[cl \mapsto s']$ and $\mathtt{P}' = \mathtt{P}[cl \mapsto \mathtt{C}']$.

Let $\mathcal{X}$ be such that $\mathcal{K}_\mathcal{X} = \mathcal{K}$, and let $T \subseteq T_\mathcal{X}$ be a set of read-only transactions in $\mathcal{X}$. It suffices to show that there exists an abstract execution $\mathcal{X}'$ such that $\mathcal{K}_{\mathcal{X}'} = \mathcal{K}'$, and

$$cl \vdash (\mathcal{X}, \mathcal{E}(cl)), \mathtt{C} \xrightarrow{(cl, T \cup \mathsf{visTx}(\mathcal{K},u), \mathcal{F})}_{(\mathsf{RP}_{\mathsf{LWW}}, \emptyset)} (\mathcal{X}', s'), \mathtt{C}'.$$

By the ASingleThread rule, we obtain

$$(\mathcal{X}, \mathcal{E})\mathtt{P}, \xrightarrow{(cl, T \cup \mathsf{visTx}(\mathcal{K},u), \mathcal{F})}_{(\mathsf{RP}_{\mathsf{LWW}}, \emptyset)} (\mathcal{X}', \mathcal{E}'), \mathtt{P}'$$

Now we perform a rule induction on the derivation of the transition

$$cl \vdash (\mathcal{K}, \mathcal{U}\,(cl)\,, \mathcal{E}(cl)), \mathtt{C} \xrightarrow{(cl,u,\mathcal{F})}_{\mathtt{ET}_\top} (\mathcal{K}', u', \sigma'), \mathtt{C}'$$

Base case: PCommit. This implies that

- $\mathtt{C} = \left[\mathtt{T}\right]$ for some $\mathtt{T}$, and $\mathtt{C}' = \mathtt{skip}$,
- $\mathcal{U}\,(cl) \sqsubseteq u$,
- let $\sigma = \mathsf{snapshot}(\mathcal{K}, u)$; then $(\mathcal{E}(cl), \sigma, \emptyset) \rightarrow^* (s', \_, \mathcal{F})$,
- $\mathcal{K}' = \mathsf{update}(\mathcal{K}, u, \mathcal{F}, t)$ for some $t \in \mathsf{nextTid}(\mathcal{K}, cl)$, and
- $\mathtt{ET}_\top \vdash (\mathcal{K}, u) \triangleright \mathcal{F} : (\mathcal{K}', u')$.

Choose an arbitrary set of of read-only transactions $T \subseteq T_\mathcal{X}$. We observe that $\mathsf{getView}(\mathcal{X}, T \cup \mathsf{visTx}(\mathcal{K}, u)) = u$ since $\mathcal{K}_\mathcal{X} = \mathcal{K}$ and Prop. 8. We can now apply Prop. 7 and ensure that $\mathsf{RP}_{\mathsf{LWW}}(\mathcal{X}, T \cup \mathsf{visTx}(\mathcal{K}, u)) = \{\sigma\}$. Let $\mathcal{X}' = \mathsf{extend}(\mathcal{X}, t, T \cup \mathsf{visTx}(\mathcal{K}, u))$. Because $\mathsf{getView}(\mathcal{X}, T \cup \mathsf{visTx}(\mathcal{K}, u)) = u$, $\mathcal{K}_\mathcal{X} = \mathcal{K}$, then by Prop. 11 we have that $\mathcal{K}_{\mathcal{X}'} = \mathsf{update}(\mathcal{K}, u, t, \mathcal{F}) = \mathcal{K}'$. To summarise, we have that $T \cup \mathsf{visTx}(\mathcal{K}, u) \subseteq T_\mathcal{X}$, $\sigma \in \mathsf{RP}_{\mathsf{LWW}}(\mathcal{X}, T \cup \mathsf{visTx}(\mathcal{K}, u))$, $(\mathcal{E}(cl), \sigma, \emptyset) \rightarrow^* (s', \_, \mathcal{F})$ and $t \in \mathsf{nextTid}(T_\mathcal{X}, cl)$. Now we can apply ACommit and infer

$$cl \vdash (\mathcal{X}, \mathcal{E}(cl)), \left[\mathtt{T}\right] \xrightarrow{cl, T \cup \mathsf{visTx}(\mathcal{K}_\mathcal{X}, u)}_{(\mathsf{RP}_{\mathsf{LWW}}, \emptyset)} (\mathcal{X}', s'), \mathtt{skip}$$

which is exactly what we wanted to prove.
Base case: PPrimitive,PChoice,PIter,PSeqSkip. These cases are trivial since they do not alter the state of $\mathcal{K}$. Inductive case: PSeq. It is derived by the I.H.

Proposition 17 (Last write win to permissive execution test). *Suppose that*

$$(\mathcal{X}, \mathcal{E}), \mathtt{P} \xrightarrow{(cl,T,\mathcal{F})}_{(\mathsf{RP}_{\mathsf{LWW}}, \emptyset)} (\mathcal{X}', \mathcal{E}'), \mathtt{P}'$$

Then for any $\mathcal{U}$ and $u \in \textsc{Views}\,(\mathcal{K}_\mathcal{X})$ such that $u \sqsubseteq \mathsf{getView}(\mathcal{X}, T)$, the following holds:

$$(\mathcal{K}_\mathcal{X}, \mathcal{U}[cl \mapsto u], \mathcal{E}), \mathtt{P} \xrightarrow{(cl, \mathsf{getView}(\mathcal{X},T), \mathcal{F})}_{\mathtt{ET}_\top} (\mathcal{K}_{\mathcal{X}'}, \mathcal{U}, \mathcal{E}'), \mathtt{P}'$$

Proof. Suppose that

$$(\mathcal{X}, \mathcal{E}), \mathtt{P} \xrightarrow{(cl,T,\mathcal{F})}_{(\mathsf{RP}_{\mathsf{LWW}},\emptyset)} (\mathcal{X}', \mathcal{E}'), \mathtt{P}'$$

Fix a function $\mathcal{U}$ from clients in $\mathrm{dom}(\mathtt{P})$ to views in $\textsc{Views}(\mathcal{K})$, and a view $u \sqsubseteq \mathsf{getView}(\mathcal{X}, T)$. We show that $(\mathcal{K}_{\mathcal{X}}, \mathcal{U}[cl \mapsto u], \mathcal{E}) \xrightarrow{(cl,\mathsf{getView}(\mathcal{X},T),\mathcal{F})}_{\mathtt{ET}_\top} (\mathcal{K}_{\mathcal{X}'}, \mathcal{U}, \mathcal{E}'), \mathtt{P}'$.

Note that the transition $\mathcal{X}, \mathcal{E}, \mathtt{P} \xrightarrow{(cl,T,\mathcal{F})}_{(\mathsf{RP}_{\mathsf{LWW}},\emptyset)} (\mathcal{X}', \mathcal{E}'), \mathtt{P}'$ can only be inferred using ASingleThread rule, from which it follows that

$$cl \vdash (\mathcal{X}, \mathcal{E}(cl)), \mathtt{P}\,(cl) \xrightarrow{(cl,T,\mathcal{F})}_{(\mathsf{RP}_{\mathsf{LWW}},\emptyset)} (\mathcal{X}', s'), \mathtt{C}'$$

for some s' such that $\mathcal{E}' = \mathcal{E}[cl \mapsto s']$ and $\mathtt{C}'$ such that $\mathtt{P}' = \mathtt{P}[cl \mapsto \mathtt{C}']$. It suffices to show that

$$cl \vdash (\mathcal{K}_{\mathcal{X}}, u, \mathcal{E}(cl)), \mathtt{P}\,(cl) \xrightarrow{(cl,\mathsf{getView}(\mathcal{K}_{\mathcal{X}},T),\mathcal{F})}_{\mathtt{ET}_\top} (\mathcal{K}_{\mathcal{X}'}, \mathcal{U}\,(cl)\,, s'), \mathtt{C}'$$

Then by applying PSingleThread we obtain

$$(\mathcal{K}_{\mathcal{X}}, \mathcal{U}[cl \mapsto u], \mathcal{E}), \mathtt{P} \xrightarrow{(cl,\mathsf{getView}(\mathcal{K}_{\mathcal{X}},T),\mathcal{F})}_{\mathtt{ET}_\top} (\mathcal{K}_{\mathcal{X}'}, \mathcal{U}, \mathcal{E}'), \mathtt{P}'$$

The rest of the proof is performed by a rule induction on the derivation to inter

$$cl \vdash (\mathcal{X}, \mathcal{E}(cl)), \mathtt{P}\,(cl) \xrightarrow{(cl,T,\mathcal{F})}_{(\mathsf{RP}_{\mathsf{LWW}},\emptyset)} (\mathcal{X}', s'), \mathtt{C}'$$

Base case: ACommit. In this case we have that

- $\mathtt{P} = \big[\mathtt{T}\big]$,
- $\mathtt{P}' = \mathtt{skip}$,
- $(\mathcal{E}(cl), \sigma, \emptyset), \mathtt{T} \rightarrow^* (s', \_, \mathcal{F}), \mathtt{skip}$ for an index $\sigma \in \mathsf{RP}_{\mathsf{LWW}}(\mathcal{X}, T)$, and
- $\mathcal{X}' = \mathsf{extend}(\mathcal{X}, t, T, \mathcal{F})$ for some $t \in \mathsf{nextTid}(\mathcal{X}, cl)$.

Furthermore, it is easy to see by induction on the length of the derivation $(\mathcal{E}(cl), \sigma, \emptyset), \mathtt{T} \rightarrow^* (s', \_, \mathcal{F}), \mathtt{skip}$, that whenever $(\mathtt{r}, k, v) \in \mathcal{F}$ then $\sigma(k) = v$. Note that $\mathsf{snapshot}(\mathcal{K}_{\mathcal{X}}, \mathsf{getView}(\mathcal{X}, T)) = \sigma$ by Prop. 7. Also, if $(\mathtt{r}, k, v) \in \mathcal{F}$ then $\sigma(k) = v$, which is possible only if $\mathcal{K}_{\mathcal{X}}(k, \max_<(\mathsf{getView}(\mathcal{X}, T)(k))) = (v, \_, \_)$. This ensures that $\mathtt{ET}_\top \vdash (\mathcal{K}_{\mathcal{X}}, \mathsf{getView}(\mathcal{X}, T)) \triangleright \mathcal{F} : \mathcal{U}\,(cl)$. We can now combine all the facts above to apply rule PCommit

$$cl \vdash (\mathcal{K}_{\mathcal{X}}, u, \mathcal{E}(cl)), \big[\mathtt{T}\big] \xrightarrow{(cl,\mathsf{getView}(\mathcal{K}_{\mathcal{X}},T),\mathcal{F})}_{\mathtt{ET}_\top} (\mathcal{K}', \mathcal{U}\,(cl)\,, s'), \mathtt{skip},$$

where $\mathcal{K}' = \mathsf{update}(\mathcal{K}_{\mathcal{X}}, t, \mathsf{getView}(\mathcal{X}, T), \mathcal{F})$. Recall that $\mathcal{X}' = \mathsf{extend}(\mathcal{X}, T, t, \mathcal{F})$. Therefore by Prop. 11 we have that $\mathcal{K}' = \mathcal{K}_{\mathcal{X}'}$, which concludes the proof of this case.

Base case: APrimitive,AChoice,AIter,ASeqSkip. These cases are trivial since they do not alter the state of $\mathcal{X}$. Inductive case: ASeq. It is derived by the I.H.

Corollary 2. *For any program* $\mathtt{P}$*,*

$$[\![\mathtt{P}]\!]_{\mathtt{ET}_\top} = \big\{\mathcal{K}_{\mathcal{X}} \;\big|\; \mathcal{X} \in [\![\mathtt{P}]\!]_{(\mathsf{RP}_{\mathsf{LWW}},\emptyset)}\big\}$$

Proof. It can be derived by Prop. 17 and Prop. 16.

E.3 Soundness and Completeness Constructor

We now show how all the results illustrated so far can be put together to show that the kv-store operational semantics is sound and complete with respect to abstract execution operational semantics.

Soundness Recall that in the abstract execution operational semantics, a client cl loses information of the visible transactions immediately after it commits a transaction. Yet such information is indirectly presented when the next transaction from the same client is committed. To define the soundness judgement (Def. 31), we introduce a notation of *invariant* (Def. 30) to encore constraints on the visible transactions after each commit.

Definition 30 (Invariant for clients). *A* client-based invariant condition*, or simply* invariant*, is a function* $I : \textsc{AbsExecs} \times \textsc{ClientID} \to \mathcal{P}(\textsc{TransID})$ *such that for any* cl *we have that* $I(\mathcal{X}, cl) \subseteq T_{\mathcal{X}}$*, and for any* cl' *such that* $cl' \neq cl$ *we have that* $I(\mathsf{extend}(\mathcal{X}, t_{cl'}, \_, \_), cl) = I(\mathcal{X}, cl)$.

Definition 31 (Soundness judgement). *An execution test* $\mathtt{ET}$ *is sound with respect to an axiomatic definition* $(\mathsf{RP}_{\mathsf{LWW}}, \mathcal{A})$ *if and only if there exists an invariant condition* I *such that if assuming that*

- *a client* cl *having an initial view* u*, commits a transaction* t *with a fingerprint* $\mathcal{F}$ *and updates the view to* u'*, which is allowed by* $\mathtt{ET}$ i.e. $\mathtt{ET} \vdash (\mathcal{K}, u) \triangleright \mathcal{F} : (\mathcal{K}', u')$ *where* $\mathcal{K}' = \mathsf{update}(\mathcal{K}, u, \mathcal{F}, t)$*,*
- *a* $\mathcal{X}$ *such that* $\mathcal{K}_{\mathcal{X}} = \mathcal{K}$ *and* $I(\mathcal{X}, cl) \subseteq \mathsf{visTx}(\mathcal{K}, u)$*,*

then there exist a set of read-only transactions T_{rd} *such that*

- *the new abstract execution* $\mathcal{X}' = \mathsf{extend}(\mathcal{X}, t, \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}}, \mathcal{F})$*,*
- *the view* u *satisfies* $\mathcal{A}$*,* i.e. $\forall \mathsf{A} \in \mathcal{A}.\ \{t' \mid (t', t) \in \mathsf{A}(\mathcal{X}')\} \subseteq \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}}$*,*
- *the invariant is preserved,* i.e. $I(\mathcal{X}', cl) \subseteq \mathsf{visTx}(\mathcal{K}', u')$.

Theorem 6 (Soundness). *If* $\mathtt{ET}$ *is sound with respect to* $(\mathsf{RP}_{\mathsf{LWW}}, \mathcal{A})$*, then*

$$\mathsf{CM}(\mathtt{ET}) \subseteq \{\mathcal{K} \mid \exists \mathcal{X} \in \mathsf{CM}(\mathsf{RP}_{\mathsf{LWW}}, \mathcal{A}))\ .\ \mathcal{K}_{\mathcal{X}} = \mathcal{K}\}$$

Proof. Let $\mathtt{ET}$ be an execution test that is sound with respect to an axiomatic definition $(\mathsf{RP}_{\mathsf{LWW}}, \mathcal{A})$. Let I be the invariant that satisfies Def. 31. Let consider an $\mathtt{ET}$-trace τ. We can assume that τ is in normal form, a trace that every view shift of a client cl is followed by a transaction from cl, and any transaction from cl must be after a view shift of cl. Without lose generality, we can also assume that the trace does not have transitions labelled as $(\_, \emptyset)$. Thus we have that the following trace τ:

$$\tau = (\mathcal{K}_0, \mathcal{U}_0) \xrightarrow{(cl_0, \varepsilon)}_{\mathtt{ET}} (\mathcal{K}_0, \mathcal{U}_0') \xrightarrow{(cl_0, \mathcal{F}_0)}_{\mathtt{ET}} (\mathcal{K}_1, \mathcal{U}_1) \xrightarrow{(cl_1, \varepsilon)}_{\mathtt{ET}} \cdots \xrightarrow{(cl_{n-1}, \mathcal{F}_{n-1})}_{\mathtt{ET}} (\mathcal{K}_n, \mathcal{U}_n)$$

For any $i : 0 \leq i \leq n$, let τ_i be the prefix of τ that contains only the first $2i$ transitions. Clearly τ_i is a valid $\mathtt{ET}$-trace, and it is also a $\mathtt{ET}_\top$-trace. By

Prop. 10, any abstract execution $\mathcal{X}_i \in \mathsf{absExec}(\tau_i)$ satisfies the last write wins policy. We show by induction on i that we can always find an abstract execution $\mathcal{X}_i \in \mathsf{absExec}(\tau_i)$ such that $\mathcal{X}_i \models \mathcal{A}$ and $I(\mathcal{X}_i, cl) \subseteq T^i_{cl}$ for any client cl and set of transactions $T^i_{cl} = \mathsf{visTx}(\mathcal{X}_i, \mathcal{U}_i(cl)) \cup T^i_{\mathsf{rd}}$, and read-only transactions T^i_{rd} in $\mathcal{X}_i$. If so, because $\mathcal{X}_i$ satisfies the last write wins policy, then it must be the case that $\mathcal{X}_i \models (\mathsf{RP}_{\mathsf{LWW}}, \mathcal{A})$. Then by choosing $i = n$, we will obtain that $\mathcal{X}_n \models (\mathsf{RP}_{\mathsf{LWW}}, \mathcal{A})$. Last, by Prop. 10, $\mathcal{K}_{\mathcal{X}_n} = \mathcal{K}_n$, and there is nothing left to prove. Now let prove such $\mathcal{X}_i \in \mathsf{absExec}(\tau_i)$ always exists.
Base case: $i = 0$. Let $\mathcal{X}_0$ be the only abstract execution included in $\mathsf{absExec}(\tau_0)$, that is $\mathcal{X}_0 = ([], \emptyset, \emptyset)$. For any $\mathsf{A} \in \mathcal{A}$, it must be the case that $\mathsf{A}(\mathcal{X}_0) \subseteq T_{\mathcal{X}_0} = \emptyset$, hence the inequation $\mathsf{A}(\mathcal{X}_0) \subseteq \mathsf{VIS}_{\mathcal{X}_0}$ is trivially satisfies. Furthermore, for the client invariant I we also require that $I(\mathcal{X}_0, \_) \subseteq T_{\mathcal{X}_0} = \emptyset$; for any client cl we can choose $T^0_{cl} = \mathsf{visTx}(\mathcal{K}_{\mathcal{X}_0}, \mathcal{U}_0(cl)) \cup \emptyset = \emptyset$. Therefore $I(\mathcal{X}_0, cl) = \emptyset \subseteq \emptyset = T^0_{cl}$.
Inductive case: $i' = i + 1$ where $i < n$. By the inductive hypothesis, there exists an abstract execution $\mathcal{X}_i$ such that

- $\mathcal{X}_i \models \mathsf{A}$ for all $\mathsf{A} \in \mathcal{A}$, and
- $I(\mathcal{X}, cl) \subseteq T^i_{cl}$ for any client cl and set of transactions $T^i_{cl} = \mathsf{visTx}(\mathcal{K}_i, \mathcal{U}_i(cl))$.

We have two transitions to check, the view shift and committing a transaction.

- the view shift transition $(\mathcal{K}_i, \mathcal{U}_i) \xrightarrow{(cl_i, \varepsilon)}_{\mathtt{ET}} (\mathcal{K}_i, \mathcal{U}'_i)$. By definition, it must be the case that $\mathcal{U}'_i = \mathcal{U}_i[cl \mapsto u'_i]$ for some u'_i such that $\mathcal{U}_i(cl) \sqsubseteq u'_i$. Let $(T^i_{cl})' = \mathsf{visTx}(\mathcal{K}_i, u'_i)$; then we have $T^i_{cl} = \mathsf{visTx}(\mathcal{K}_i, \mathcal{U}_i(cl)) \subseteq \mathsf{visTx}(\mathcal{K}_i, u'_i) = (T^i_{cl})'$ As a consequence, $I(\mathcal{X}, cl) \subseteq T^i_{cl} \subseteq (T^i_{cl})'$.
- the commit transaction transition $(\mathcal{K}_i, \mathcal{U}'_i) \xrightarrow{(cl_i, \mathcal{F}_i)}_{\mathtt{ET}} (\mathcal{K}_{i+1}, \mathcal{U}_{i+1})$. A necessary condition for this transition to appear in τ is that $\mathtt{ET} \vdash (\mathcal{K}_i, \mathcal{U}(cl)) \triangleright \mathcal{F}_i : (\mathcal{K}_{i+1}, \mathcal{U}_{i+1}(cl))$. Because I is the invariant to derive that $\mathtt{ET}$ is sound with respect to $\mathcal{A}$, and because $I(\mathcal{X}_i, cl_i) \subseteq (T^i_{cl})'$, then by Def. 31 we have the following:
  - there exists a set of read-only transactions T_{rd} such that

$$\left\{t' \;\middle|\; (t', t_{(cl,i)}) \in \mathsf{A}(\mathcal{X}_{i+1})\right\} \subseteq {T^i_{cl}}' \cup T_{\mathsf{rd}}$$

    where $t_{(cl,i)} \in \mathsf{nextTid}(\mathcal{K}_i, cl)$ and $\mathcal{X}_{i+1} = \mathsf{extend}\big(\mathcal{X}_i, t_{(cl,i)}, (T^i_{cl})' \cup T_{\mathsf{rd}}, \mathcal{F}_i\big)$,
  - $I(\mathcal{X}_{i+1}, cl) \subseteq \mathsf{visTx}(\mathcal{K}_{i+1}, \mathcal{U}_{i+1}(cl))$.

  Because $\mathcal{X}_i \in \mathsf{absExec}(\tau_i)$, by definition of $\mathsf{absExec}(\_)$ we have that $\mathcal{X}_{i+1} \in \mathsf{absExec}(\tau)$ (under the assumption that $\mathcal{F}_i \neq \emptyset$), and because $\mathsf{lastConf}(\tau_{i+1}) = (\mathcal{K}_{i+1}, \_)$, then $\mathcal{K}_{\mathcal{X}_{i+1}} = \mathcal{K}_{i+1}$.
  Now we need to check if $\mathcal{X}_{i+1}$ satisfies $\mathcal{A}$ and the invariant I is preserved.
  - $\mathsf{A}(\mathcal{X}_{i+1}) \subseteq \mathsf{VIS}^{i+1}_{\mathcal{X}}$ for any $\mathsf{A} \in \mathcal{A}$. Fix $\mathsf{A} \in \mathcal{A}$ and $(t', t) \in \mathsf{A}(\mathcal{X}_{i+1})$. Because $\mathcal{X}_{i+1} = \mathsf{extend}\big(\mathcal{X}_i, t_{(cl,i)}, (T^i_{cl})' \cup T_{\mathsf{rd}}, \mathcal{F}_i\big)$, we distinguish between two cases.
    * If $t = t_{(cl,i)}$, then it must be the case that $t' \in (T^i_{cl})' \cup T_{\mathsf{rd}}$, and by definition of extend we have that $(t', t_{(cl,i)}) \in \mathsf{VIS}_{\mathcal{X}_{i+1}}$.

  - If $t \neq t_{(cl,i)}$, then we have that $t, t' \in T_{\mathcal{X}_i}$. Because $\mathcal{X}_i$ and $\mathcal{X}_{i+1}$ agree on $T_{\mathcal{X}_i}$, then $(t', t) \in \mathsf{A}(\mathcal{X}_i)$. Because $\mathcal{X}_i \models \mathsf{A}$, then $(t', t) \in \mathsf{VIS}_{\mathcal{X}_i}$. By definition of extend, it follows that $(t', t) \in \mathsf{VIS}_{\mathcal{X}_{i+1}}$.
- Finally, we show the invariant is preserved. Fix a client cl'.
  - If $cl' = cl$, then we have already proved that $I(\mathcal{X}_{i+1}, cl) \subseteq T^{i+1}_{cl}$.
  - if $cl' \neq cl$, then note that $\mathcal{U}_i(cl') = \mathcal{U}'_i(cl') = \mathcal{U}_{i+1}(cl')$, and in particular $(T^{cl'}_i)' = \mathsf{visTx}\big(\mathcal{X}_i, \mathcal{U}'_i(cl')\big) = \mathsf{visTx}\big(\mathcal{X}_{i+1}, \mathcal{U}_{i+1}(cl')\big) = T^{i+1}_{cl'}$. By the inductive hypothesis we know that $I(\mathcal{X}_i, cl) \subseteq T^i_{cl'}$, and by the definition of invariant, we have $I(\mathcal{X}_{i+1}, cl) \subseteq T^i_{cl'} = T^{i+1}_{cl'}$.

Corollary 3. *If* $\mathtt{ET}$ *is sound with respect to* $(\mathsf{RP}_{\mathsf{LWW}}, \mathcal{A})$*, then for any program* $\mathtt{P}$*,* $[\![\mathtt{P}]\!]_{\mathtt{ET}} \subseteq \big\{\mathcal{K}_{\mathcal{X}} \;\big|\; \mathcal{X} \in [\![\mathtt{P}]\!]_{(\mathsf{RP}_{\mathsf{LWW}}, \mathcal{A})}\big\}$.

Proof.

$$
\begin{aligned}
[\![\mathtt{P}]\!]_{\mathtt{ET}} &= [\![\mathtt{P}]\!]_{\mathtt{ET}_\top} \cap \mathsf{CM}(\mathtt{ET}) \\
&= \{\mathcal{K}_{\mathcal{X}} \mid \mathcal{X} \text{ satisfies } \mathsf{RP}_{\mathsf{LWW}}\} \cap \mathsf{CM}(\mathtt{ET}) \\
&\overset{Theorem\ 6}{\subseteq} \{\mathcal{K}_{\mathcal{X}} \mid \mathcal{X} \text{ satisfies } \mathsf{RP}_{\mathsf{LWW}} \wedge \mathcal{X} \in \mathsf{CM}(\mathsf{RP}_{\mathsf{LWW}}, \mathcal{A})\} \\
&= \big\{\mathcal{K}_{\mathcal{X}} \;\big|\; \mathcal{X} \in [\![\mathtt{P}]\!]_{(\mathsf{RP}_{\mathsf{LWW}}, \mathcal{A})}\big\}
\end{aligned}
$$

Completeness The Completeness judgement is in Def. 32. Given a transaction t_i from client cl, it converts the visible transactions $\mathsf{VIS}^{-1}_{\mathcal{X}}(t_i)$ into view and such view should satisfy the $\mathtt{ET}$. Note that $\mathcal{X}$ does not contain precise information about final view after update, yet the visible transactions of the immediate next transaction from the same client cl include those information.

Definition 32. *An execution test* $\mathtt{ET}$ *is* complete *with respect to an axiomatic definition* $(\mathsf{RP}_{\mathsf{LWW}}, \mathcal{A})$ *if, for any abstract execution* $\mathcal{X} \in \mathsf{CM}(\mathsf{RP}_{\mathsf{LWW}}, \mathcal{A})$ *and index* $i : 1 \leq i < |T_{\mathcal{X}}|$ *such that* $t_i \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t_{i+1}$*, there exist an initial view* u_i *and a final view* u'_i *where*

- $u_i = \mathsf{getView}\big(\mathcal{X}, \mathsf{VIS}^{-1}_{\mathcal{X}}(t_i)\big)$,
- *let* $t_i = t^n_{cl}$ *for some* cl, n*;*
  - *if the transaction* $t'_i = \min_{\mathsf{SO}_{\mathcal{X}}} \Big\{t' \;\Big|\; t_i \xrightarrow{\mathsf{SO}_{\mathcal{X}}} t'\Big\}$ *is defined, then* $u' = \mathsf{getView}(\mathcal{X}, T_i)$ *where* $T_i \subseteq (\mathsf{AR}^{-1}_{\mathcal{X}})^?(t_i) \cap \mathsf{VIS}^{-1}_{\mathcal{X}}(t'_i)$*;*
  - *otherwise* $u' = \mathsf{getView}(\mathcal{X}, T_i)$ *where* $T_i \subseteq (\mathsf{AR}^{-1}_{\mathcal{X}})^?(t_i)$,
- $\mathtt{ET} \vdash (\mathcal{K}_{\mathsf{cut}(\mathcal{X}, i-1)}, u_i) \triangleright \mathscr{T}_{\mathcal{X}}(t_i) : (\mathcal{K}_{\mathsf{cut}(\mathcal{X}, i)}, u'_i)$.

Theorem 7. *Let* $\mathtt{ET}$ *be an execution test that is complete with respect to an axiomatic definition* $(\mathsf{RP}_{\mathsf{LWW}}, \mathcal{A})$*. Then* $\mathsf{CM}(\mathsf{RP}_{\mathsf{LWW}}, \mathcal{A}) \subseteq \mathsf{CM}(\mathtt{ET})$.

Proof. Fix an abstract execution $\mathcal{X} \in \mathsf{CM}(\mathsf{RP}_{\mathsf{LWW}}, \mathcal{A})$. For any $i : 1 \leq i < |T_{\mathcal{X}}|$, suppose that t_i that is the i-*th* transaction follows the arbitrary order, i.e. $t_i \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t_{i+1}$ and let cl_i be the client of the i-*th* step, i.e. $t_i = t^{\cdot}_{cl_i}$. Because $\mathtt{ET}$ is complete with respect to $(\mathsf{RP}_{\mathsf{LWW}}, \mathcal{A})$, for any step i we can find an initial views u_i,and a final view u'_i such that

- $u_i = \mathsf{getView}\big(\mathcal{X}, \mathsf{VIS}_{\mathcal{X}}^{-1}(t_i)\big)$,
- there exists a set of transactions T_i such that $\mathsf{getView}(\mathcal{X}, T_i) = u_i'$, and either $\mathsf{min}_{\mathsf{SO}_{\mathcal{X}}} \left\{ t' \;\middle|\; t_i \xrightarrow{\mathsf{SO}_{\mathcal{X}}} t' \right\}$ is is defined and $T_i \subseteq (\mathsf{AR}_{\mathcal{X}}^{-1})^?(t_i) \cap \mathsf{VIS}_{\mathcal{X}}^{-1}(t')$, or $T_i \subseteq (\mathsf{AR}_{\mathcal{X}}^{-1})^?(t_i)$,
- $\mathtt{ET} \vdash (\mathcal{K}_{\mathsf{cut}(\mathcal{X},i-1)}, u_i) \triangleright \mathscr{T}_{\mathcal{X}}(t_i) : (\mathcal{K}_{\mathsf{cut}(\mathcal{X},i)}, u_i')$.

Given above, let $\mathcal{K}_i = \mathsf{cut}(\mathcal{X}, i)$ and $\mathcal{F}_i = \mathscr{T}_{\mathcal{X}}(t_i)$. Define the views for clients as

$$\mathcal{U}_0 = \lambda cl \in \left\{ cl' \;\middle|\; \exists t \in T_{\mathcal{X}}.\ t = t_{cl'} \right\}. \lambda k. \{0\} \quad \mathcal{U}_{i-1}' = \mathcal{U}_i[cl_i \mapsto u_i] \quad \mathcal{U}_i = \mathcal{U}_{i-1}'[cl_i \mapsto u_i']$$

and the ke-stores as

$$\mathcal{K}_0 = \lambda k.(v_0, t_0, \emptyset) \quad \mathcal{K}_i = \mathsf{update}(\mathcal{K}_{i-1}, u_i, \mathcal{F}_i, t_i)$$

Now by Prop. 13 we have that the following sequence of $\mathtt{ET}_\top$-reductions

$$(\mathcal{K}_0, \mathcal{U}_0) \xrightarrow{(cl_1,\varepsilon)}_{\mathtt{ET}_\top} (\mathcal{K}_0, \mathcal{U}_0') \xrightarrow{(cl_1,\mathcal{F}_1)}_{\mathtt{ET}_\top} (\mathcal{K}_1, \mathcal{U}_1) \xrightarrow{(cl_2,\varepsilon)}_{\mathtt{ET}_\top} \cdots \xrightarrow{(cl_n,\mathcal{F}_n)}_{\mathtt{ET}_\top} (\mathcal{K}_n, \mathcal{U}_n)$$

Note that $\mathcal{K}_i = \mathcal{K}_{\mathsf{cut}(\mathcal{X},i)}$. Because $\mathtt{ET} \vdash (\mathcal{K}_{\mathsf{cut}(\mathcal{X},i-1)}, u_i) \triangleright \mathcal{F}_i : (\mathcal{K}_i, u_i')$, or equivalently $\mathtt{ET} \vdash (\mathcal{K}_{\mathsf{cut}(\mathcal{X},i-1)}, \mathcal{U}_{i-1}'(cl_i)) \triangleright \mathcal{F}_i : (\mathcal{K}_{\mathsf{cut}(\mathcal{X},i-1)}, \mathcal{U}_i(cl_i))$, therefore

$$(\mathcal{K}_0, \mathcal{U}_0) \xrightarrow{(cl_1,\varepsilon)}_{\mathtt{ET}} (\mathcal{K}_0, \mathcal{U}_0') \xrightarrow{(cl_1,\mathcal{F}_1)}_{\mathtt{ET}} (\mathcal{K}_1, \mathcal{U}_1) \xrightarrow{(cl_2,\varepsilon)}_{\mathtt{ET}} \cdots \xrightarrow{(cl_n,\mathcal{F}_n)}_{\mathtt{ET}} (\mathcal{K}_n, \mathcal{U}_n)$$

It follows that $\mathcal{K}_n \in \mathsf{CM}(\mathtt{ET})$ then $\mathcal{K}_n = \mathcal{K}_{\mathsf{cut}(\mathcal{X},n)} = \mathcal{K}_{\mathcal{X}}$, and there is nothing left to prove.

Corollary 4. *If* $\mathtt{ET}$ *is complete with respect to* $(\mathsf{RP}_{\mathsf{LWW}}, \mathcal{A})$*, then for any program* $\mathtt{P}$*,*

$$\left\{ \mathcal{K}_{\mathcal{X}} \;\middle|\; \mathcal{X} \in [\![\mathtt{P}]\!]_{(\mathsf{RP}_{\mathsf{LWW}},\mathcal{A})} \right\} \subseteq [\![\mathtt{P}]\!]_{\mathtt{ET}}$$

Proof.

$$\begin{aligned}
\left\{ \mathcal{K}_{\mathcal{X}} \;\middle|\; \mathcal{X} \in [\![\mathtt{P}]\!]_{(\mathsf{RP}_{\mathsf{LWW}},\mathcal{A})} \right\} &= \{\mathcal{K}_{\mathcal{X}} \mid \mathcal{X} \text{ satisfies } \mathsf{RP}_{\mathsf{LWW}} \wedge \mathcal{X} \in \mathsf{CM}(\mathsf{RP}_{\mathsf{LWW}}, \mathcal{A})\} \\
&\overset{Theorem\ 7}{\subseteq} \{\mathcal{K}_{\mathcal{X}} \mid \mathcal{X} \text{ satisfies } \mathsf{RP}_{\mathsf{LWW}}\} \cap \mathsf{CM}(\mathtt{ET}) \\
&= [\![\mathtt{P}]\!]_{\mathtt{ET}_\top} \cap \mathsf{CM}(\mathtt{ET}) \\
&= [\![\mathtt{P}]\!]_{\mathtt{ET}}
\end{aligned}$$

F The Soundness and Completeness of Execution Tests

We use Defs. 31 and 32 to prove the soundness and completeness of execution tests with respect to axiomatic definitions. It is sufficient to match these two definition, then by Cors. 3 and 4 we have $\mathsf{CM}(\mathtt{ET}) = \{\mathcal{K}_\mathcal{X} \mid \mathcal{X} \in \mathsf{CM}(\mathsf{RP}_{\mathsf{LWW}}, \mathcal{A})\}$.

We first prove the Theorem 8, which states that the least fix point of view matches the constraint on the visibility relation on abstract execution.

Theorem 8 (View closure to visibility closure). *Assume $\mathcal{K}$ and $\mathcal{X}$ such that $\mathcal{K} = \mathcal{K}_\mathcal{X}$, and $R_\mathcal{K}$ and $R_\mathcal{X}$ such that $R_\mathcal{K} = R_\mathcal{X}$. For any $t, \mathcal{F}$, if there is a view $u = \mathsf{getView}\left(\mathcal{K}, \left(R_\mathcal{K}^{-1}\right)^* (\mathsf{visTx}(\mathcal{K}, u))\right)$, then the new abstract execution $\mathcal{X}' = \mathsf{extend}\left(\mathcal{X}, t, \mathcal{F}, \left(R_\mathcal{K}^{-1}\right)^* (\mathsf{visTx}(\mathcal{K}, u))\right)$ satisfies $R_\mathcal{X}^{-1}(\mathsf{VIS}_{\mathcal{X}'}^{-1}(t)) \subseteq \mathsf{VIS}_{\mathcal{X}'}^{-1}(t)$. Conversely, If there a new abstract execution $\mathcal{X}' = \mathsf{extend}(\mathcal{X}, t, \mathcal{F}, T)$ for some T that satisfies $R_\mathcal{X}^{-1}(T) \subseteq T$, and if a view $u = \mathsf{getView}(\mathcal{K}, T)$, then the view $u = \mathsf{getView}\left(\mathcal{K}, \left(R_\mathcal{K}^{-1}\right)^* (\mathsf{visTx}(\mathcal{K}, u))\right)$.*

Proof. Assume $\mathcal{K}$ and $\mathcal{X}$ such that $\mathcal{K} = \mathcal{K}_\mathcal{X}$, and $R_\mathcal{K}$ and $R_\mathcal{X}$ such that $R_\mathcal{K} = R_\mathcal{X}$, Assume $t, \mathcal{F}$. Let $T = \left(R_\mathcal{K}^{-1}\right)^* (\mathsf{visTx}(\mathcal{K}, u))$. Assume that a view satisfies $u = \mathsf{getView}(\mathcal{K}, T)$. By the definition of extend, the visible transactions $\mathsf{VIS}_{\mathcal{X}'}^{-1}(t) = T$. Let consider transactions t', t'' such that $t' \xrightarrow{R_\mathcal{X}} t'' \xrightarrow{\mathsf{VIS}_{\mathcal{X}'}} t$. This means there exists a natural number n such that $t'' \in \left(R_\mathcal{K}^{-1}\right)^n (\mathsf{visTx}(\mathcal{K}, u))$. Given that $R_\mathcal{K} = R_\mathcal{X}$, it follows $t' \in \left(R_\mathcal{K}^{-1}\right)^{n+1} (\mathsf{visTx}(\mathcal{K}, u))$, then $t' \in T$ and so $t' \xrightarrow{\mathsf{VIS}_{\mathcal{X}'}} t$.

Assume there a new abstract execution $\mathcal{X}' = \mathsf{extend}(\mathcal{X}, t, \mathcal{F}, T)$, that satisfies $R_\mathcal{X}^{-1}(T) \subseteq T$. Assume $u = \mathsf{getView}(\mathcal{K}, T)$. Note that $R_\mathcal{X} = R_\mathcal{K}$. It suffices to prove $\{t' \in T \mid t' \text{ has writes}\} = \left\{t' \in \left(R_\mathcal{K}^{-1}\right)^* (\mathsf{visTx}(\mathcal{K}, u)) \;\middle|\; t' \text{ has writes}\right\}$.

- Assume a transaction $t' \in T$ that has writes. It is easy to see there are k, i such that $i \in u\,(k)$ and $\mathsf{w}(\mathcal{K}(k, i)) \in T$. This means $t' \in \mathsf{visTx}(\mathcal{K}, u)$.
- Assume a transaction $t' \in \mathsf{visTx}(\mathcal{K}, u)$, we now prove $\left(R_\mathcal{K}^{-1}\right)^n (t') \subseteq T$ for all n.
  - Base case: n = 0. It trivially holds that $t' \in \mathsf{visTx}(\mathcal{K}, u) \subseteq T$.
  - Inductive case: n + 1. Assume a transaction $t''' \in \left(R_\mathcal{K}^{-1}\right)^{n+1} (t')$. It means there is a $t'' \in \left(R_\mathcal{K}^{-1}\right)^n (t')$ such that $t''' \xrightarrow{R_\mathcal{K}} t''$. By I.H., $t'' \in T$. Given $R_\mathcal{K} = R_\mathcal{X}$ and $R_\mathcal{X}^{-1}(T) \subseteq T$, it is known that $t''' \in T$.

F.1 Monotonic Read MR

The execution test $\mathtt{ET}_{\mathtt{MR}}$ is sound with respect to the axiomatic definition [44]

$$(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.\mathsf{VIS}_\mathcal{X}; \mathsf{SO}_\mathcal{X}\})$$

We choose an invariant as the following,

$$I(\mathcal{X}, cl) = \left(\bigcup_{\left\{ t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N} \right\}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl}) \right) \setminus T_{\mathsf{rd}}$$

where T_{rd} is all the read-only transactions in $\bigcup_{\left\{ t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N} \right\}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl})$. Assume a kv-store $\mathcal{K}$, an initial and a final view u, u' a fingerprint $\mathcal{F}$ such that $\mathtt{ET}_{\mathtt{MR}} \vdash (\mathcal{K}, u) \triangleright \mathcal{F} : (\mathcal{K}', u')$. Also choose an arbitrary cl, a transaction identifier $t \in \mathsf{nextTid}(\mathcal{K}, cl)$, and an abstract execution $\mathcal{X}$ such that $\mathcal{K}_{\mathcal{X}} = \mathcal{K}$ and

$$I(\mathcal{X}, cl) \subseteq \mathsf{visTx}(\mathcal{K}, u) \tag{6.1}$$

Let $\mathcal{X}' = \mathsf{extend}(\mathcal{X}, t, \mathcal{F}, \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}})$. We now check if $\mathcal{X}'$ satisfies the axiomatic definition and the invariant is preserved:

- $\{t' \mid (t', t) \in \mathsf{VIS}_{\mathcal{X}'}; \mathsf{SO}_{\mathcal{X}'}\} \subseteq \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}}$. Suppose that $t' \xrightarrow{\mathsf{VIS}_{\mathcal{X}'}} t'' \xrightarrow{\mathsf{SO}_{\mathcal{X}'}} t$ for some t', t''. We show that $t' \in I(\mathcal{X}, cl)$, and then Eq. (6.1) ensures that $t' \in \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}}$. Suppose $t'' \xrightarrow{\mathsf{SO}_{\mathcal{X}'}} t$, then $t'' = t^n_{cl}$ for some $n \in \mathbb{N}$. Because $t'' \neq t$ and $T_{\mathcal{X}'} \setminus T_{\mathcal{X}} = \{t\}$, we also have that $t'' \in \mathcal{X}$. By the invariant of $I(\mathcal{X}, cl)$, we have that $\mathsf{VIS}^{-1}_{\mathcal{X}}(cl) \subseteq I(\mathcal{X}, cl)$: because $t' \xrightarrow{\mathsf{VIS}_{\mathcal{X}'}} t''$ and $t'' \neq t$ we have that $t' \xrightarrow{\mathsf{VIS}_{\mathcal{X}}} t''$ and therefore $t' \in I(\mathcal{X}, cl)$.
- $I(\mathcal{X}', cl) \subseteq \mathsf{visTx}(\mathcal{X}', u') = \mathsf{visTx}(\mathcal{K}', u')$. In this case, because $\mathtt{ET}_{\mathtt{MR}} \vdash (\mathcal{K}, u) \triangleright \mathcal{F} : (\mathcal{K}', u')$, then it must be the case that $u \sqsubseteq u'$. A trivial consequence of this fact is that $\mathsf{visTx}(\mathcal{K}, u) \subseteq \mathsf{visTx}(\mathcal{K}, u')$. Also, because $\mathcal{X}' = \mathsf{extend}(\mathcal{X}, t, \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}})$, we have that $\mathsf{visTx}(\mathcal{K}_{\mathcal{X}}, u) = \mathsf{visTx}(\mathcal{K}_{\mathcal{X}'}, u)$. Finally, note that $\{t^n_{cl} \in \mathcal{X}' \mid n \in \mathbb{N}\} = \{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\} \cup t$, that for any $t^n_{cl} \in T_{\mathcal{X}}$ we have that $\mathsf{VIS}^{-1}_{\mathcal{X}'}(t^n_{cl}) = \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl})$, and that $\mathsf{VIS}^{-1}_{\mathcal{X}'}(t) = \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}}$. Using all these facts, we obtain

$$\begin{aligned}
I(\mathcal{X}', cl) &= \left(\bigcup_{\left\{ t^n_{cl} \in \mathcal{X}' \mid n \in \mathbb{N} \right\}} \mathsf{VIS}^{-1}_{\mathcal{X}'}(t^n_{cl}) \right) \setminus T_{\mathsf{rd}} \\
&= \left(\left(\bigcup_{\left\{ t^n_{cl} \in \mathcal{X} \mid n \in \mathbb{N} \right\}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl}) \right) \setminus T_{\mathsf{rd}} \right) \cup \left(\mathsf{VIS}^{-1}_{\mathcal{X}'}(t) \setminus T_{\mathsf{rd}} \right) \\
&= I(\mathcal{X}, cl) \cup \mathsf{visTx}(\mathcal{K}, u) \\
&\overset{(6.1)}{\subseteq} \mathsf{visTx}(\mathcal{K}, u) \\
&= \mathsf{visTx}(\mathcal{K}_{\mathcal{X}}, u) \\
&= \mathsf{visTx}(\mathcal{K}_{\mathcal{X}'}, u) \\
&\subseteq \mathsf{visTx}(\mathcal{K}_{\mathcal{X}'}, u')
\end{aligned}$$

We show that the execution test $\mathtt{ET}_{\mathtt{MR}}$ is complete with respect to the axiomatic definition

$$(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.(\mathsf{VIS}_{\mathcal{X}}; \mathsf{SO}_{\mathcal{X}})\})$$

Let $\mathcal{X}$ be an abstract execution that satisfies the definition $\mathsf{CM}(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.(\mathsf{VIS}_{\mathcal{X}}; \mathsf{SO}_{\mathcal{X}})\})$, and consider a transaction $t \in T_{\mathcal{X}}$. Assume i-th transaction t_i in the arbitrary order, and let $u_i = \mathsf{getView}\big(\mathcal{X}, \mathsf{VIS}_{\mathcal{X}}^{-1}(t_i)\big)$. We have two possible cases:

- the transaction $t'_i = \mathsf{min}_{\mathsf{SO}_{\mathcal{X}}} \left\{ t' \;\middle|\; t_i \xrightarrow{\mathsf{SO}_{\mathcal{X}}} t' \right\}$ is defined. In this case let

  $$u'_i = \mathsf{getView}\big(\mathcal{X}, (\mathsf{AR}_{\mathcal{X}}^{-1})^{?}(t_i) \cap \mathsf{VIS}_{\mathcal{X}}^{-1}(t'_i)\big)$$

  Note that $t_i \xrightarrow{\mathsf{SO}_{\mathcal{X}}} t'_i$, and because $\mathcal{X} \models \mathsf{VIS}_{\mathcal{X}}; \mathsf{SO}_{\mathcal{X}}$, it follows that $\mathsf{VIS}_{\mathcal{X}}^{-1}(t_i) \subseteq \mathsf{VIS}_{\mathcal{X}}^{-1}(t'_i)$. We also have that $\mathsf{VIS}_{\mathcal{X}}^{-1}(t_i) \subseteq (\mathsf{AR}_{\mathcal{X}}^{-1})^{?}(t_i)$ because of the definition of abstract execution. It follows that

  $$\mathsf{VIS}_{\mathcal{X}}^{-1}(t_i) \subseteq (\mathsf{AR}_{\mathcal{X}}^{-1})^{?}(t_i) \cap \mathsf{VIS}_{\mathcal{X}}^{-1}(t'_i),$$

  Recall that $u_i = \mathsf{getView}\big(\mathcal{X}, \mathsf{VIS}_{\mathcal{X}}^{-1}(t_i)\big)$, and $u'_i = \mathsf{getView}\big(\mathcal{X}, (\mathsf{AR}_{\mathcal{X}}^{-1})^{?}(t_i) \cap \mathsf{VIS}_{\mathcal{X}}^{-1}(t'_i)\big)$. Thus we have that $u_i \sqsubseteq u'_i$, and therefore $\mathtt{ET}_{\mathtt{MR}} \vdash (\mathcal{K}_{\mathsf{cut}(\mathcal{X},i)}, u_i) \triangleright \mathscr{T}_{\mathcal{X}}(t_i) : (\mathcal{K}_{\mathsf{cut}(\mathcal{X},i+1)}, u'_i)$.
- the transaction $t'_i = \mathsf{min}_{\mathsf{SO}_{\mathcal{X}}} \left\{ t' \;\middle|\; t_i \xrightarrow{\mathsf{SO}_{\mathcal{X}}} t_i \right\}$ is not defined. In this case, let

  $$u'_i = \mathsf{getView}\big(\mathcal{X}, (\mathsf{AR}_{\mathcal{X}}^{-1})^{?}(t_i)\big)$$

  As for the case above, we have that $u_i \sqsubseteq u'_i$, and therefore $\mathtt{ET}_{\mathtt{MR}} \vdash (\mathcal{K}_{\mathsf{cut}(\mathcal{X},i)}, u_i) \triangleright \mathscr{T}_{\mathcal{X}}(t_i) : (\mathcal{K}_{\mathsf{cut}(\mathcal{X},i+1)}, u'_i, u'_i)$.

F.2 Monotonic Write MW

The execution test $\mathtt{ET}_{\mathtt{MW}}$ is sound with respect to the axiomatic definition [44]

$$(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.(\mathsf{SO}_{\mathcal{X}} \cap \mathsf{WW}_{\mathcal{X}}); \mathsf{VIS}_{\mathcal{X}}\})$$

We pick the invariant as empty set given the fact of no constraint on the view after update:

$$I(\mathcal{X}, cl) = \emptyset$$

Assume a kv-store $\mathcal{K}$, an initial and a final view u, u' a fingerprint $\mathcal{F}$ such that $\mathtt{ET}_{\mathtt{MW}} \vdash (\mathcal{K}, u) \triangleright \mathcal{F} : (\mathcal{K}', u')$. Also choose an arbitrary cl, a transaction identifier $t \in \mathsf{nextTid}(\mathcal{K}, cl)$, and an abstract execution $\mathcal{X}$ such that $\mathcal{K}_{\mathcal{X}} = \mathcal{K}$ and $I(\mathcal{X}, cl) = \emptyset \subseteq \mathsf{visTx}(\mathcal{K}, u)$. Note that since the invariant is empty set, it remains to prove that there exists a set of read-only transactions T_{rd} such that $\mathcal{X}' = \mathsf{extend}(\mathcal{X}, t, \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}}, \mathcal{F})$ and:

$$\forall t'.\ (t', t) \in (\mathsf{SO}_{\mathcal{X}'} \cap \mathsf{WW}_{\mathcal{X}'}); \mathsf{VIS}_{\mathcal{X}'} \Rightarrow t' \in \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}}$$

which can be derived from Theorem 8.

The execution test $\mathtt{ET}_{\mathtt{MW}}$ is complete with respect to the axiomatic definition

$$(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.((\mathsf{SO}_{\mathcal{X}} \cap \mathsf{WW}_{\mathcal{X}}); \mathsf{VIS}_{\mathcal{X}})\})$$

Let $\mathcal{X}$ be an abstract execution that satisfies the definition $\mathsf{CM}(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.((\mathsf{SO}_{\mathcal{X}} \cap \mathsf{WW}_{\mathcal{X}}); \mathsf{VIS}_{\mathcal{X}})\})$, and consider a transaction $t \in T_{\mathcal{X}}$. Let $\mathcal{K} = \mathcal{K}_{\mathcal{X}}$. Assume i-*th* transaction t_i in the arbitrary order, and let view $u_i = \mathsf{getView}\big(\mathcal{K}, \mathsf{VIS}_{\mathcal{X}}^{-1}(t_i)\big)$. We also pick any final view such that $u_i' \subseteq \mathsf{getView}\big(\mathcal{K}, (\mathsf{AR}_{\mathcal{X}}^{-1})^?(t_i)\big)$. It suffices to prove $\mathtt{ET}_{\mathtt{MW}} \vdash (\mathcal{K}_{\mathsf{cut}(\mathcal{X},i-1)}, u_i) \triangleright \mathscr{T}_{\mathcal{X}}(t_i) : (\mathcal{K}_{\mathsf{cut}(\mathcal{X},i-1)}, u_i')$. It means to prove the following:

$$u_i = \mathsf{getView}\Big(\mathcal{K}_{\mathsf{cut}(\mathcal{X},i-1)}, \mathsf{lfpTx}\Big(\mathcal{K}_{\mathsf{cut}(\mathcal{X},i-1)}, u, \mathsf{SO} \cap \mathsf{WW}_{\mathcal{K}_{\mathsf{cut}(\mathcal{X},i-1)}}\Big)\Big)$$

which can be derived from Theorem 8.

F.3 Read Your Write RYW

The execution test $\mathtt{ET}_{\mathtt{RYW}}$ is sound with respect to the axiomatic definition [44] $(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.\,\mathsf{SO}_{\mathcal{X}}\})$. We pick an invariant for the $\mathtt{ET}_{\mathtt{RYW}}$ as the following:

$$I(\mathcal{X}, cl) = \left(\bigcup_{\left\{t_{cl}^n \in T_{\mathcal{X}} \;\middle|\; n \in \mathbb{N}\right\}} (\mathsf{SO}_{\mathcal{X}}^{-1})^?(t_{cl}^n) \right) \setminus T_{\mathsf{rd}}$$

where T_{rd} is all the read-only transactions in $\bigcup_{\left\{t_{cl}^n \in T_{\mathcal{X}} \;\middle|\; n \in \mathbb{N}\right\}} (\mathsf{SO}_{\mathcal{X}}^{-1})^?(t_{cl}^n)$. Assume a kv-store $\mathcal{K}$, an initial and a final view u, u' a fingerprint $\mathcal{F}$ such that $\mathtt{ET}_{\mathtt{RYW}} \vdash (\mathcal{K}, u) \triangleright \mathcal{F} : (\mathcal{K}', u')$. Also choose an arbitrary cl, a transaction identifier $t_{cl}^n \in \mathsf{nextTid}(\mathcal{K}, cl)$, and an abstract execution $\mathcal{X}$ such that $\mathcal{K}_{\mathcal{X}} = \mathcal{K}$ and $I(\mathcal{X}, cl) \subseteq \mathsf{visTx}(\mathcal{K}, u)$. Let a new abstract execution $\mathcal{X}' = \mathsf{extend}(\mathcal{X}, t_{cl}^n, \mathcal{F}, \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}})$. We need to prove that $\mathcal{X}'$ satisfies the constraint and the invariant is preserved:

- $t \in \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}}$ for all t such that $t \xrightarrow{\mathsf{SO}_{\mathcal{X}'}} t_{cl}^n$. Assume a transaction t such that $t \xrightarrow{\mathsf{SO}_{\mathcal{X}'}} t_{cl}^n$. It immediately implies that $t = t_{cl}^m$ where $m < n$ and $t_{cl}^m \in \mathcal{X}$. Thus we prove that

$$t \in \left(\bigcup_{\left\{t_{cl}^n \in T_{\mathcal{X}} \;\middle|\; n \in \mathbb{N}\right\}} (\mathsf{SO}_{\mathcal{X}}^{-1})^?(t_{cl}^n) \right) \subseteq \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}}$$

- $I(\mathcal{X}', cl) \subseteq \mathsf{visTx}(\mathcal{K}_{\mathcal{X}'}, u')$. Let $T_{\mathsf{rd}}' = T_{\mathsf{rd}}$ if the new transaction t_{cl}^n has writes, otherwise $T_{\mathsf{rd}}' = T_{\mathsf{rd}} \cup \{t_{cl}^n\}$. First we have

$$I(\mathcal{X}', cl) = \left(\bigcup_{\left\{t_{cl}^m \in T_{\mathcal{X}'} \;\middle|\; m \in \mathbb{N}\right\}} (\mathsf{SO}_{\mathcal{X}'}^{-1})^?(t_{cl}^m) \right) \setminus T_{\mathsf{rd}}' = \big((\mathsf{SO}_{\mathcal{X}'}^{-1})^?(t_{cl}^n)\big) \setminus T_{\mathsf{rd}}'$$

Note that t^n_{cl} is the latest transaction committed by the client cl. For any transaction $t \in (\mathsf{SO}^{-1}_{\mathcal{X}'})^?(t^n_{cl}) \setminus T'_{\mathsf{rd}}$ that has write, because execution test requires $z \in u'(k)$ for any key k and index z such that $\mathsf{w}(\mathcal{K}_{\mathcal{X}'}(k,z)) \xrightarrow{\mathsf{SO}_{\mathcal{X}}} t$, then $t \in \mathsf{visTx}(\mathcal{K}_{\mathcal{X}'}, u')$ as what we wanted.

The execution test $\mathtt{ET}_{\mathtt{RYW}}$ is complete with respect to the axiomatic definition $(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.\mathsf{SO}_{\mathcal{X}}\})$. Let $\mathcal{X}$ be an abstract execution that satisfies the definition $\mathsf{CM}(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.\mathsf{SO}_{\mathcal{X}}\})$. Assume i-$th$ transaction t_i in the arbitrary order, and let view $u_i = \mathsf{getView}\big(\mathcal{X}, \mathsf{VIS}^{-1}_{\mathcal{X}}(t_i)\big)$. We construct the final view u'_i depending on whether t_i is the last transaction from the client.

- If the transaction $t'_i = \mathrm{min}_{\mathsf{SO}_{\mathcal{X}}}\left(\left\{t' \;\middle|\; t_i \xrightarrow{\mathsf{SO}_{\mathcal{X}}} t'\right\}\right)$ is defined, then $u'_i = \mathsf{getView}(\mathcal{X}, T_i)$ where $T_i \subseteq (\mathsf{AR}^{-1}_{\mathcal{X}})^?(t_i) \cap \mathsf{VIS}^{-1}_{\mathcal{X}}(t'_i)$ for some T_i. Given the definition $\lambda\mathcal{X}.\mathsf{SO}_{\mathcal{X}}$, we know $\mathsf{SO}^{-1}_{\mathcal{X}}(t'_i) \subseteq \mathsf{VIS}^{-1}_{\mathcal{X}}(t'_i)$, so $(\mathsf{AR}^{-1}_{\mathcal{X}})^?(t_i) \cap \mathsf{SO}^{-1}(t'_i) = (\mathsf{SO}^{-1})^?(t_i) \subseteq T_i$. Take j, k such that $\mathsf{w}\big(\mathcal{K}_{\mathsf{cut}(\mathcal{X},i)}(k,j)\big) \xrightarrow{\mathsf{SO}^?} t'_i$. By the constraint of $\mathcal{X}$, that is $\mathsf{SO}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$, it follows $\mathsf{w}\big(\mathcal{K}_{\mathsf{cut}(\mathcal{X},i)}(k,j)\big) \in T_i$. Recall $u'_i = \mathsf{getView}\big(\mathcal{K}_{\mathsf{cut}(\mathcal{X},i)}, T_i\big)$. By the definition of $\mathsf{getView}$, it follows $i \in u'_i(k)$. Therefore $\mathtt{ET}_{\mathtt{RYW}} \vdash (\mathcal{K}_{\mathsf{cut}(\mathcal{X},i-1)}, u_i) \triangleright \mathscr{T}_{\mathcal{X}}(t_i) : (\mathcal{K}_{\mathsf{cut}(\mathcal{X},i)}, u'_i)$.
- If there is no other transaction after t_i from the same client, we pick $u'_i = \mathsf{getView}(\mathcal{X}, T_i)$ where $T_i = (\mathsf{SO}^{-1}_{\mathcal{X}})^?(t_i)$, so $\mathtt{ET}_{\mathtt{RYW}} \vdash (\mathcal{K}_{\mathsf{cut}(\mathcal{X},i-1)}, u_i) \triangleright \mathscr{T}_{\mathcal{X}}(t_i) : (\mathcal{K}_{\mathsf{cut}(\mathcal{X},i)}, u'_i)$.

F.4 Write Following Read `WFR`

The write-read relation on $\mathcal{X}$ is defined as the following:

$$\mathsf{WR}\,(\mathcal{X}, k) \triangleq \left\{(t, t') \;\middle|\; \exists v.\ (\mathtt{w}, k, v) \in_{\mathcal{X}} t \wedge (\mathtt{r}, k, v) \in_{\mathcal{X}} t' \wedge t = \max_{\mathsf{AR}\left(\mathsf{VIS}^{-1}(t')\right)}\right\}$$

The notation $\mathsf{WR}_{\mathcal{X}}$ is defined as $\mathsf{WR}_{\mathcal{X}} \triangleq \bigcup_{k \in \mathrm{KEYS}} \mathsf{WR}\,(\mathcal{X}, k)$. Note that for a kv-store $\mathcal{K}$ such that $\mathcal{K} = \mathcal{K}_{\mathcal{X}}$, by the definition of $\mathcal{K} = \mathcal{K}_{\mathcal{X}}$, the following holds:

$$\mathsf{WR}_{\mathcal{X}} = \{(t, t') \mid \exists k, i.\ \mathcal{K}(k, i) = (\_, t, t' \cup \_)\}$$

Note that such $\mathsf{WR}_{\mathcal{X}}$ coincides with $\mathsf{WR}_{\mathcal{G}}$ and $\mathsf{WR}_{\mathcal{K}}$.

The execution test $\mathtt{ET}_{\mathtt{WFR}}$ is sound with respect to the axiomatic definition [44]

$$(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.\mathsf{WR}_{\mathcal{X}}; (\mathsf{SO} \cap \mathsf{RW}_{\mathcal{X}})^?; \mathsf{VIS}_{\mathcal{X}}\})$$

We pick the invariant as $I(\mathcal{X}, cl) = \emptyset$, given the fact of no constraint on the view after update. Assume a kv-store $\mathcal{K}$, an initial and a final view u, u' a fingerprint $\mathcal{F}$ such that $\mathtt{ET}_{\mathtt{WFR}} \vdash (\mathcal{K}, u) \triangleright \mathcal{F} : (\mathcal{K}', u')$. Also choose an arbitrary cl, a transaction identifier $t \in \mathsf{nextTid}(\mathcal{K}, cl)$, and an abstract execution $\mathcal{X}$ such that $\mathcal{K}_{\mathcal{X}} = \mathcal{K}$ and $I(\mathcal{X}, cl) = \emptyset \subseteq \mathsf{visTx}(\mathcal{K}, u)$. Note that since the invariant is empty set, it remains to prove there is a set of read-only transactions T_{rd} such that Let $\mathcal{X}' = \mathsf{extend}(\mathcal{X}, t, \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}}, \mathcal{F})$ and

$$\forall t'.\ (t', t) \in \mathsf{WR}_{\mathcal{X}'}; \mathsf{SO}^?_{\mathcal{X}'}; \mathsf{VIS}_{\mathcal{X}'} \Rightarrow t' \in \mathsf{visTx}(\mathcal{K}, u)$$

which can be derived from Theorem 8.

The execution test $\mathtt{ET}_{\mathtt{WFR}}$ is complete with respect to the axiomatic definition

$$(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.\mathsf{WR}; \mathsf{WR}_{\mathcal{K}}; (\mathsf{SO} \cap \mathsf{RW}_{\mathcal{X}'})^?; \mathsf{VIS}_{\mathcal{X}'}\})$$

Assume i-*th* transaction t_i in the arbitrary order, and let view $u_i = \mathsf{getView}\big(\mathcal{K}_{\mathsf{cut}(\mathcal{X},i-1)}, \mathsf{VIS}^{-1}_{\mathcal{X}}(t_i)\big)$. We also pick any final view such that $u'_i \subseteq \mathsf{getView}\big(\mathcal{K}_{\mathsf{cut}(\mathcal{X},i)}, (\mathsf{AR}^{-1}_{\mathcal{X}})^?(t_i)\big)$. Note that there is nothing to prove for u'_i, so it is sufficient to prove the following:

$$u_i = \mathsf{getView}\Big(\mathcal{K}_{\mathsf{cut}(\mathcal{X},i-1)}, \mathsf{lfpTx}\Big(\mathcal{K}_{\mathsf{cut}(\mathcal{X},i-1)}, u, \mathsf{WR}_{\mathcal{K}_{\mathsf{cut}(\mathcal{X},i-1)}}; \mathsf{SO}\Big)\Big)$$

which can be derived from Theorem 8.

F.5 Causal Consistency CC

The wildly used definition on abstract executions for causal consistency is that VIS is transitive. Yet it is for the sack of elegant definition, while there is a minimum visibility relation given by $(\mathsf{WR}_{\mathcal{X}} \cup \mathsf{SO}_{\mathcal{X}}); \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$ (Lemma 7).

Lemma 7. *For any abstract execution $\mathcal{X}$ under last-write-win, if it satisfies the following:*

$$(\mathsf{WR}_{\mathcal{X}} \cup \mathsf{SO}_{\mathcal{X}}); \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}} \quad \mathsf{SO}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$$

There exists a new abstract execution $\mathcal{X}'$ where $T_{\mathcal{X}} = T_{\mathcal{X}'}$, $\mathsf{AR}_{\mathcal{X}} = \mathsf{AR}_{\mathcal{X}'}$, $\mathsf{VIS}_{\mathcal{X}'}; \mathsf{VIS}_{\mathcal{X}'} \subseteq \mathsf{VIS}_{\mathcal{X}'}$, and under last-write-win $\mathscr{T}_{\mathcal{X}}(t) = \mathscr{T}_{\mathcal{X}'}(t)$ for all transactions t.

Proof. To recall, the write-read relation under a key $\mathsf{WR}\,(\mathcal{X}, k)$ is defined as $\mathsf{WR}\,(\mathcal{X}, k) \triangleq \Big\{(t, t') \;\Big|\; \exists v.\ (\mathtt{w}, k, v) \in_{\mathcal{X}} t \wedge (\mathtt{r}, k, v) \in_{\mathcal{X}} t' \wedge t = \max_{\mathsf{AR}(\mathsf{VIS}^{-1}(t'))}\Big\}$. Given an $\mathcal{X}$ that satisfies the following

$$(\mathsf{WR}_{\mathcal{X}} \cup \mathsf{SO}_{\mathcal{X}}); \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}} \quad \mathsf{SO}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$$

we erase some visibility relation for each transaction following the order of arbitration AR until the visibility is transitive. Assume the i-*th* transaction t_i with respect to the arbitration order. Let R_i denote a new visibility for transaction t_i such that $R_i\lfloor_2 = \{t_i\}$ and the visibility relation before (including) t_i is transitive. Let $\mathcal{X}_i = \mathcal{K}_{\mathsf{cut}(\mathcal{X},i)}$ and $\mathsf{VIS}_i = \bigcup_{0 \le k \le i} R_i$. For each step, says i-*th* step, we preserve the following:

$$\mathsf{VIS}_i; \mathsf{VIS}_i \subseteq \mathsf{VIS}_i \tag{6.2}$$

$$\forall t.\ (t, t_i) \in R_i \Rightarrow (t, t_i) \in (\mathsf{WR}_i \cup \mathsf{SO}_i) \tag{6.3}$$

- Base case: $i = 1$ and $R_1 = \emptyset$. Assume it is from client cl. There is no transaction committed before, so $\mathsf{VIS}_1 = \emptyset$ and $\mathsf{VIS}_1; \mathsf{VIS}_1 \subseteq \mathsf{VIS}_1$ as Eq. (6.2).
- Inductive case: i-*th* step. Suppose the (i-1)-*th* step satisfies Eq. (6.2) and Eq. (6.3). Let consider i-*th* step and the transaction t_i. Initially we take R_i as empty set. We first extend R_i by closing with respect to WR_i and prove that it does not affect any read from the transaction t_i. Then we will do the same for SO_i.

- WR_i. For any read $(\mathtt{r}, k, v) \in t_i$, there must be a transaction t_j that $t_j \xrightarrow{\mathsf{WR}(\mathcal{X}_i,k),\mathsf{AR}} t_i$ and $j < i$. We include $(t_j, t_i) \in R_i$. Let consider all the visible transactions of t_j. Assume a transaction $t' \in \mathsf{VIS}_{i-1}^{-1}(t_j)$, thus $t' \in \mathsf{VIS}_j^{-1}(t_j) = R_j^{-1}(t_j)$. It is safe to include $(t', t_i) \in R_i$ without affecting the read result, because those transaction t' is already visible for t_i in the abstract execution $\mathcal{X}$: by Eq. (6.3) we know $R_j \subseteq (\mathsf{WR}_j \cup \mathsf{SO}_j)^+ \subseteq (\mathsf{WR}_\mathcal{X} \cup \mathsf{SO}_\mathcal{X})^+$, and by the definition of $\mathsf{WR}\,(\mathcal{X}_i, k)$ we know $\mathsf{WR}\,(\mathcal{X}_i, k) \subseteq \mathsf{VIS}_\mathcal{X}$.
- Given $\mathsf{SO}_\mathcal{X} \subseteq \mathsf{VIS}_\mathcal{X}$, we include (t_j, t_i) for some t_j such that $t_j \xrightarrow{\mathsf{SO}_\mathcal{X}} t_i$. For the similar reason as WR, it is safe to includes all the visible transactions t' for t_j, i.e. $t' \in R_j^{-1}$.

By the construction, both Eq. (6.2) and Eq. (6.3) are preserved. Thus we have the proof.

Proposition 18. *For any abstract execution $\mathcal{X}$ under last-write-win, if it satisfies the following:*

$$(\mathsf{WR}_\mathcal{X} \cup \mathsf{SO}_\mathcal{X})^+; \mathsf{VIS}_\mathcal{X} \subseteq \mathsf{VIS}_\mathcal{X} \quad \mathsf{SO}_\mathcal{X} \subseteq \mathsf{VIS}_\mathcal{X}$$

then

$$\exists R \subseteq \mathsf{AR}_\mathcal{X}.\ \mathsf{VIS} = (\mathsf{WR}_\mathcal{X} \cup \mathsf{SO}_\mathcal{X} \cup R)^+$$

By Lemma 7, the execution test $\mathtt{ET}_{\mathtt{CC}}$ is sound with respect to the axiomatic definition

$$(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.(\mathsf{SO}_\mathcal{X} \cup \mathsf{WR}_\mathcal{X})^+; \mathsf{VIS}_\mathcal{X}, \lambda\mathcal{X}.\,\mathsf{SO}_\mathcal{X}\})$$

We pick an invariant for the $\mathtt{ET}_{\mathtt{CC}}$ as the union of those for `MR` and `RYW` shown in the following:

$$I_1(\mathcal{X}, cl) = \left(\bigcup_{\{t_{cl}^n \in T_\mathcal{X} \mid n \in \mathbb{N}\}} \mathsf{VIS}_\mathcal{X}^{-1}(t_{cl}^n) \right) \setminus T_{\mathsf{rd}}$$

$$I_2(\mathcal{X}, cl) = \left(\bigcup_{\{t_{cl}^n \in T_\mathcal{X} \mid n \in \mathbb{N}\}} (\mathsf{SO}_\mathcal{X}^{-1})^?(t_{cl}^n) \right) \setminus T_{\mathsf{rd}}$$

where T_{rd} is all the read-only transactions included in both:

$$T_{\mathsf{rd}} \in \left(\bigcup_{\{t_{cl}^n \in T_\mathcal{X} \mid n \in \mathbb{N}\}} \mathsf{VIS}_\mathcal{X}^{-1}(t_{cl}^n) \right)$$

and

$$T_{\mathsf{rd}} \in \left(\bigcup_{\{t_{cl}^n \in T_\mathcal{X} \mid n \in \mathbb{N}\}} (\mathsf{SO}_\mathcal{X}^{-1})^?(t_{cl}^n) \right)$$

Assume a kv-store $\mathcal{K}$, an initial and a final view u, u' a fingerprint $\mathcal{F}$ such that $\mathtt{ET}_{\mathtt{CC}} \vdash (\mathcal{K}, u) \triangleright \mathcal{F} : (\mathcal{K}', u')$. Also choose an arbitrary cl, a transaction identifier $t^n_{cl} \in \mathsf{nextTid}(\mathcal{K}, cl)$, and an abstract execution $\mathcal{X}$ such that $\mathcal{K}_{\mathcal{X}} = \mathcal{K}$ and $I_1(\mathcal{X}, cl) \cup I_2(\mathcal{X}, cl) \subseteq \mathsf{visTx}(\mathcal{K}, u)$. We are about to prove there exists an extra set of read-only transactions T'_{rd} such that the new abstract execution $\mathcal{X}' = \mathsf{extend}(\mathcal{X}, t^n_{cl}, \mathcal{F}, \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}} \cup T'_{\mathsf{rd}})$ and:

$$\forall t.\ (t, t^n_{cl}) \in \mathsf{SO}_{\mathcal{X}'} \Rightarrow t \in \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}} \cup T'_{\mathsf{rd}} \tag{6.4}$$

$$\forall t.\ (t, t^n_{cl}) \in (\mathsf{SO}_{\mathcal{X}'} \cup \mathsf{WR}_{\mathcal{X}'}); \mathsf{VIS}_{\mathcal{X}'} \Rightarrow t \in \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}} \cup T'_{\mathsf{rd}} \tag{6.5}$$

$$I_1(\mathcal{X}', cl) \cup I_2(\mathcal{X}', cl) \subseteq \mathsf{visTx}(\mathcal{K}_{\mathcal{X}'}, u') \tag{6.6}$$

- The invariant I_2 implies Eq. (6.4) as the same as `RYW` in §F.3.
- Eq. (6.5). Note that $(t, t^n_{cl}) \in (\mathsf{SO}_{\mathcal{X}'} \cup \mathsf{WR}_{\mathcal{X}'}); \mathsf{VIS}_{\mathcal{X}'} \Rightarrow (t, t^n_{cl}) \in (\mathsf{SO}_{\mathcal{X}} \cup \mathsf{WR}_{\mathcal{X}}); \mathsf{VIS}_{\mathcal{X}'}$. Also, recall that $\mathsf{SO}_{\mathcal{X}} = \mathsf{SO}_{\mathcal{K}}$ and $\mathsf{WR}_{\mathcal{X}} = \mathsf{WR}_{\mathcal{K}}$. Let $T'_{\mathsf{rd}} = \mathsf{lfpTx}(\mathcal{K}, u, \mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}})$. This means that $\mathcal{X}' = \mathsf{extend}(\mathcal{X}, t^n_{cl}, \mathcal{F}, \mathsf{lfpTx}(\mathcal{K}, u, \mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}}) \cup T_{\mathsf{rd}})$.
  Let assume $t \xrightarrow{\mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}}} t'$ and $t' \in \mathsf{lfpTx}(\mathcal{K}, u, \mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}}) \cup T_{\mathsf{rd}}$. We have two possible cases:
  - If $t' \in \mathsf{lfpTx}(\mathcal{K}, u, \mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}})$, by Theorem 8, we know $t \in \mathsf{lfpTx}(\mathcal{K}, u, \mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}})$.
  - If $t' \in T_{\mathsf{rd}}$, there are two cases:
    * $t' \in \left(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl})\right)$. By the property of $\mathcal{X}$ (before update) that $(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}}); \mathsf{VIS}_{\mathcal{X}} \in \mathsf{VIS}_{\mathcal{X}}$, it is known that $t \in \left(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl})\right)$, that is, $t \in \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}}$.
    * $t' \in \left(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{SO}^{-1}_{\mathcal{X}}(t^n_{cl})\right)$. Then we know $t \in (\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}})^{-1} \left(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{SO}^{-1}_{\mathcal{X}}(t^n_{cl})\right)$. By the property of $\mathcal{X}$ (before update) that $\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}} \in \mathsf{VIS}_{\mathcal{X}}$, it follows:

$$\begin{aligned} t &\in VIS^{-1}_{\mathcal{X}} \left(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{SO}^{-1}_{\mathcal{X}}(t^n_{cl}) \right) \\ &= \left(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl}) \right) \\ &= \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}} \end{aligned}$$

- Finally the new abstract execution preserves the invariant I_1 and I_2 because `CC` satisfies `MW` and `RYW`. The proofs are the same as those in §F.1 and §F.3.

The execution test $\mathtt{ET}_{\mathtt{CC}}$ is complete with respect to the axiomatic definition

$$(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.\mathsf{VIS}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}}, \lambda\mathcal{X}.\mathsf{SO}_{\mathcal{X}}\})$$

Assume i-*th* transaction t_i in the arbitrary order, and let view $u_i = \mathsf{getView}\big(\mathcal{X}, \mathsf{VIS}^{-1}_{\mathcal{X}}(t_i)\big)$. We pick final view as $u'_i = \mathsf{getView}\big(\mathcal{X}, (\mathsf{AR}^{-1}_{\mathcal{X}})^?(t_i) \cap \mathsf{VIS}^{-1}_{\mathcal{X}}(t'_i)\big)$, if $t'_i = \min_{\mathsf{SO}} \left\{ t' \;\middle|\; t_i \xrightarrow{\mathsf{SO}} t' \right\}$ is defined, otherwise $u'_i = \mathsf{getView}\big(\mathcal{X}, (\mathsf{AR}^{-1}_{\mathcal{X}})^?(t_i)\big)$. Let the $\mathcal{K} = \mathcal{K}_{\mathsf{cut}(\mathcal{X}, i-1)}$. Now we prove the three parts separately.

- $\texttt{MR}$. By Prop. 18 since $\mathsf{VIS}_{\mathcal{X}}; \mathsf{SO}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$ so it follows as in §F.1.
- $\texttt{RYW}$. For $\texttt{RYW}$, since $\mathsf{WR}_{\mathcal{X}}; \mathsf{SO}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$, the proof is as the same proof as in §F.3.
- $\texttt{allowed}(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO})$. It is derived from Theorem 8 and $(\mathsf{WR}_{\mathcal{X}} \cup \mathsf{SO}); \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$.

F.6 Update Atomic $\texttt{UA}$

Given abstract execution $\mathcal{X}$, we define write-write relation for a key k as the following [15]:

$$\mathsf{WW}\,(\mathcal{X}, k) \triangleq \left\{ (t, t') \;\middle|\; t \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t' \wedge (\mathtt{w}, k, \_) \in t \wedge (\mathtt{w}, k, \_) \in t' \right\}$$

Then, the notation $\mathsf{WW}_{\mathcal{X}} \triangleq \bigcup_{k \in \mathrm{KEYS}} \mathsf{WW}\,(\mathcal{X}, k)$. Note that for a kv-store $\mathcal{K}$ such that $\mathcal{K} = \mathcal{K}_{\mathcal{X}}$, by the definition of $\mathcal{K} = \mathcal{K}_{\mathcal{X}}$, the following holds:

$$\mathsf{WW}_{\mathcal{X}} = \{(t, t') \mid \exists k, i, j.\ t = \mathsf{w}(\mathcal{K}(k, i)) \wedge t' = \mathsf{w}(\mathcal{K}(k, j)) \wedge i < j\}$$

Also the $\mathsf{WW}_{\mathcal{X}}$ coincides with $\mathsf{WW}_{\mathcal{G}}$ and $\mathsf{WW}_{\mathcal{K}}$.

The execution test $\mathtt{ET}_{\mathtt{UA}}$ is sound with respect to the axiomatic definition $(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.\mathsf{WW}_{\mathcal{X}}\})$. We pick the invariant as $I(\mathcal{X}, cl) = \emptyset$, given the fact of no constraint on the final view. Assume a kv-store $\mathcal{K}$, an initial and a final view u, u' a fingerprint $\mathcal{F}$ such that $\mathtt{ET}_{\mathtt{UA}} \vdash (\mathcal{K}, u) \triangleright \mathcal{F} : (\mathcal{K}', u')$. Also choose an arbitrary cl, a transaction identifier $t \in \mathsf{nextTid}(\mathcal{K}, cl)$, and an abstract execution $\mathcal{X}$ such that $\mathcal{K}_{\mathcal{X}} = \mathcal{K}$ and $I(\mathcal{X}, cl) = \emptyset \subseteq \mathsf{visTx}(\mathcal{K}, u)$. Let $\mathcal{X}' = \mathsf{extend}(\mathcal{X}, t, \mathsf{visTx}(\mathcal{K}, u), \mathcal{F})$. Note that since the invariant is empty set, it remains to prove the following:

$$\forall t'.\ t' \xrightarrow{\mathsf{WW}_{\mathcal{X}'}} t \Rightarrow t' \in \mathsf{visTx}(\mathcal{K}, u)$$

Assume a transaction t' that writes to a key k as t, i.e. $t' \xrightarrow{\mathsf{WW}_{\mathcal{X}'}} t$. Since that t' is a transaction already existing in $\mathcal{K}$, we have $\mathsf{w}(\mathcal{K}(k, i)) = t'$ for some index i and key k. It means $(\mathtt{w}, k, \mathsf{val}(\mathcal{K}(k, i))) \in \mathcal{F}$. By the execution test of $\mathtt{UA}$, we know $i \in u\,(k)$ therefore $t' \in \mathsf{visTx}(\mathcal{K}, u)$.

The execution test $\mathtt{ET}_{\mathtt{UA}}$ is complete with respect to the axiomatic definition $(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.\mathsf{WW}_{\mathcal{X}}\})$. Assume i-$th$ transaction t_i in the arbitrary order, and let view $u_i = \mathsf{getView}\big(\mathcal{X}, \mathsf{VIS}_{\mathcal{X}}^{-1}(t_i)\big)$. We also pick any final view such that $u'_i \subseteq \mathsf{getView}\big(\mathcal{X}, (\mathsf{AR}_{\mathcal{X}}^{-1})^?(t_i)\big)$. Note that there is nothing to prove for u'_i, so it is sufficient to prove the following:

$$\forall k.\ (\mathtt{w}, k, \_) \in \mathscr{T}_{\mathcal{X}}(t_i) \Rightarrow \forall j : 0 \leq j < |\mathcal{K}_{\mathsf{cut}(\mathcal{X}, i-1)}(k)|.\ j \in u_i(k)$$

Let consider a key k that have been overwritten by the transaction t_i. By the constraint of $\mathcal{X}$ that $\mathsf{WW}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$, for any transaction t that writes to the same key k and committed before t_i, they are included in the visible set $t \in \mathsf{VIS}_{\mathcal{X}}^{-1}(t_i)$. Note that $t \xrightarrow{\mathsf{WW}_{\mathcal{X}}} t_i \Rightarrow t \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t_i \Rightarrow t \in \mathcal{K}_{\mathsf{cut}(\mathcal{X}, i-1)}$. Since that the transaction t write to the key k, it means $\mathsf{w}(\mathcal{K}_{\mathsf{cut}(\mathcal{X}, i-1)}(k, j)) = t$ for some index j. Then by the definition of $\mathsf{getView}$, we have $j \in u_i(k)$.

F.7 Consistency Prefix CP

Given abstract execution $\mathcal{X}$, we define read-write read-write relation:

$$\mathsf{RW}\left(\mathcal{X}, k\right) \triangleq \left\{(t, t') \;\middle|\; t \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t' \wedge (\mathtt{r}, k, \_) \in t \wedge (\mathtt{w}, k, \_) \in t'\right\}$$

It is easy to see $\mathsf{RW}\left(\mathcal{X}, k\right)$ can be derived from $\mathsf{WW}\left(\mathcal{X}, k\right)$ and $\mathsf{WR}\left(\mathcal{X}, k\right)$ as the following:

$$\mathsf{RW}\left(\mathcal{X}, k\right) = \{(t, t') \mid \exists t''.\ (t'', t) \in \mathsf{WR}\left(\mathcal{X}, k\right) \wedge (t'', t') \in \mathsf{WW}\left(\mathcal{X}, k\right)\}$$

Then, the notation $\mathsf{RW}_{\mathcal{X}} \triangleq \bigcup_{k \in \mathrm{KEYS}} \mathsf{RW}\left(\mathcal{X}, k\right)$. Note that for a kv-store $\mathcal{K}$ such that $\mathcal{K} = \mathcal{K}_{\mathcal{X}}$, by the definition of $\mathcal{K} = \mathcal{K}_{\mathcal{X}}$, the following holds:

$$\mathsf{RW}_{\mathcal{X}} = \{(t, t') \mid \exists k, i, j.\ t \in \mathsf{rs}(\mathcal{K}(k, i)) \wedge t' = \mathsf{w}(\mathcal{K}(k, j)) \wedge i < j\}$$

The $\mathsf{RW}_{\mathcal{X}}$ also coincides with $\mathsf{RW}_{\mathcal{G}}$ and $\mathsf{RW}_{\mathcal{K}}$.

An abstract execution $\mathcal{X}$ satisfies consistency prefix (CP), if it satisfies $\mathsf{AR}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$ and $\mathsf{SO}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$. Given the definition, there is a corresponding definition on dependency graph by solve the following inequalities:

$$\begin{array}{l}
\mathsf{WR} \subseteq \mathsf{VIS} \\
\mathsf{WW} \subseteq \mathsf{AR} \\
\mathsf{VIS} \subseteq \mathsf{AR} \\
\mathsf{VIS}; \mathsf{RW} \subseteq \mathsf{AR} \\
\mathsf{AR}; \mathsf{AR} \subseteq \mathsf{AR} \\
\mathsf{SO} \subseteq \mathsf{VIS} \\
\mathsf{AR}; \mathsf{VIS} \subseteq \mathsf{VIS}
\end{array}$$

By solving the inequalities the visibility and arbitration relations are:

$$\begin{aligned}
\mathsf{AR} &\triangleq \left((\mathsf{SO} \cup \mathsf{WR}); \mathsf{RW}^{?} \cup \mathsf{WW} \cup R\right)^{+} \\
\mathsf{VIS} &\triangleq \left((\mathsf{SO} \cup \mathsf{WR}); \mathsf{RW}^{?} \cup \mathsf{WW} \cup R\right)^{*}; (\mathsf{SO} \cup \mathsf{WR})
\end{aligned}$$

for some relation $R \subseteq \mathsf{AR}$. When $R = \emptyset$, it is the smallest solution therefore the minimum visibility required.

Lemma 8. *For any abstract execution $\mathcal{X}$, if it satisfies*

$$\left((\mathsf{SO} \cup \mathsf{WR}); \mathsf{RW}^{?} \cup \mathsf{WW}\right); \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}} \qquad \mathsf{SO}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$$

then there exists a new $\mathcal{X}'$ such that $T_{\mathcal{X}} = T_{\mathcal{X}'}$, under last-write-win $\mathscr{T}_{\mathcal{X}}(t) = \mathscr{T}_{\mathcal{X}'}(t)$ for all transactions t, and the relations satisfy the following:

$$\mathsf{AR}_{\mathcal{X}'}; \mathsf{VIS}_{\mathcal{X}'} \subseteq \mathsf{VIS}_{\mathcal{X}'} \qquad \mathsf{SO}_{\mathcal{X}'} \subseteq \mathsf{VIS}_{\mathcal{X}'}$$

and vice versa.

Proof. Assume abstract execution $\mathcal{X}'$ that satisfies $\mathsf{AR}_{\mathcal{X}'}; \mathsf{VIS}_{\mathcal{X}'} \subseteq \mathsf{VIS}_{\mathcal{X}'}$ and $\mathsf{SO}_{\mathcal{X}'} \subseteq \mathsf{VIS}_{\mathcal{X}'}$. We already show that:

$$\mathsf{AR}_{\mathcal{X}'} = \Big((\mathsf{SO}_{\mathcal{X}} \cup \mathsf{WR}_{\mathcal{X}}); \mathsf{RW}^{?}_{\mathcal{X}} \cup \mathsf{WW}_{\mathcal{X}} \cup R\Big)^{+}$$
$$\mathsf{VIS}_{\mathcal{X}'} = \Big((\mathsf{SO}_{\mathcal{X}} \cup \mathsf{WR}_{\mathcal{X}}); \mathsf{RW}^{?}_{\mathcal{X}} \cup \mathsf{WW}_{\mathcal{X}} \cup R\Big)^{*}; (\mathsf{SO}_{\mathcal{X}} \cup \mathsf{WR}_{\mathcal{X}})$$

for some relation $R \subseteq \mathsf{AR}_{\mathcal{X}'}$. If we take $R = \emptyset$, we have the proof for:

$$\mathsf{SO} \subseteq \mathsf{VIS}_{\mathcal{X}} \qquad \Big((\mathsf{SO}_{\mathcal{X}} \cup \mathsf{WR}_{\mathcal{X}}); \mathsf{RW}^{?}_{\mathcal{X}} \cup \mathsf{WW}_{\mathcal{X}}\Big); \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$$

For another way, we pick the R that extends $\Big((\mathsf{SO}_{\mathcal{X}} \cup \mathsf{WR}_{\mathcal{X}}); \mathsf{RW}^{?}_{\mathcal{X}} \cup \mathsf{WW}_{\mathcal{X}} \cup R\Big)^{+}$ to a total order.

By Lemma 8 to prove soundness and completeness of $\mathtt{ET}_{\mathtt{CP}}$, it is sufficient to use the definition:

$$(\mathsf{RP}_{\mathsf{LWW}}, \Big\{\lambda\mathcal{X}.\ \Big((\mathsf{SO} \cup \mathsf{WR}); \mathsf{RW}^{?} \cup \mathsf{WW}\Big); \mathsf{VIS}_{\mathcal{X}}, \lambda\mathcal{X}.\ \mathsf{SO}_{\mathcal{X}}\Big\})$$

For the soundness, we pick the invariant as the following:

$$I_1(\mathcal{X}, cl) = \left(\bigcup_{\left\{t^i_{cl} \in T_{\mathcal{X}} \;\middle|\; i \in \mathbb{N}\right\}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^i_{cl})\right) \setminus T_{\mathsf{rd}}$$
$$I_2(\mathcal{X}, cl) = \left(\bigcup_{\left\{t^i_{cl} \in T_{\mathcal{X}} \;\middle|\; i \in \mathbb{N}\right\}} (\mathsf{SO}^{-1}_{\mathcal{X}})^{?}(t^i_{cl})\right) \setminus T_{\mathsf{rd}}$$

where T_{rd} is all the read-only transactions included in both

$$T_{\mathsf{rd}} \in \left(\bigcup_{\left\{t^i_{cl} \in T_{\mathcal{X}} \;\middle|\; i \in \mathbb{N}\right\}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^i_{cl})\right)$$

and

$$T_{\mathsf{rd}} \in \left(\bigcup_{\left\{t^i_{cl} \in T_{\mathcal{X}} \;\middle|\; i \in \mathbb{N}\right\}} (\mathsf{SO}^{-1}_{\mathcal{X}})^{?}(t^i_{cl})\right)$$

Assume a key-value store $\mathcal{K}$, an initial and a final view u, u' a fingerprint $\mathcal{F}$ such that $\mathtt{ET}_{\mathtt{CP}} \vdash (\mathcal{K}, u) \triangleright \mathcal{F} : (\mathcal{K}', u')$. Also choose an arbitrary cl, a transaction identifier $t^n_{cl} \in \mathsf{nextTid}(\mathcal{K}, cl)$, and an abstract execution $\mathcal{X}$ such that $\mathcal{K}_{\mathcal{X}} = \mathcal{K}$ and $I_1(\mathcal{X}, cl) \cup I_2(\mathcal{X}, cl) \subseteq \mathsf{visTx}(\mathcal{K}, u)$. We are about to prove that there exists

an extra set of read-only transaction T'_{rd} such that the new abstract execution $\mathcal{X}' = \mathsf{extend}(\mathcal{X}, t^n_{cl}, \mathcal{F}, \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}} \cup T'_{\mathsf{rd}})$ and

$$\forall t.\ (t, t^n_{cl}) \in \mathsf{SO}_{\mathcal{X}'} \Rightarrow t \in \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}} \cup T'_{\mathsf{rd}} \tag{6.7}$$

$$\forall t.\ (t, t^n_{cl}) \in \left((\mathsf{SO}_{\mathcal{X}'} \cup \mathsf{WR}_{\mathcal{X}'}); \mathsf{RW}^?_{\mathcal{X}'} \cup \mathsf{WW}_{\mathcal{X}'}\right); \mathsf{VIS}_{\mathcal{X}'} \Rightarrow t \in \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}} \cup T'_{\mathsf{rd}} \tag{6.8}$$

$$I_1(\mathcal{X}', cl) \cup I_2(\mathcal{X}', cl) \subseteq \mathsf{visTx}(\mathcal{K}_{\mathcal{X}'}, u') \tag{6.9}$$

- the invariant I_2 implies the Eq. (6.7) where the proof is the same as `RYW` in §F.3.
- Eq. (6.8). Note that

$$\begin{aligned}(t, t^n_{cl}) \in \left((\mathsf{SO}_{\mathcal{X}'} \cup \mathsf{WR}_{\mathcal{X}'}); \mathsf{RW}^?_{\mathcal{X}'} \cup \mathsf{WW}_{\mathcal{X}'}\right); \mathsf{VIS}_{\mathcal{X}'} \\ \Rightarrow (t, t^n_{cl}) \in \left((\mathsf{SO}_{\mathcal{X}} \cup \mathsf{WR}_{\mathcal{X}}); \mathsf{RW}^?_{\mathcal{X}} \cup \mathsf{WW}_{\mathcal{X}}\right); \mathsf{VIS}_{\mathcal{X}'}\end{aligned}$$

  Also, recall that $R_{\mathcal{X}} = R_{\mathcal{K}}$ for $R \in \{\mathsf{SO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}\}$. Let

$$T'_{\mathsf{rd}} = \mathsf{lfpTx}\Big(\mathcal{K}, u, (\mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}}); \mathsf{RW}^?_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}}\Big)$$

  Let assume $t \xrightarrow{(\mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}}); \mathsf{RW}^?_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}}} t'$ and $t' \in \mathsf{lfpTx}\Big(\mathcal{K}, u, (\mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}}); \mathsf{RW}^?_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}}\Big) \cup T_{\mathsf{rd}}$. We have two possible cases:
  - If $t' \in \mathsf{lfpTx}\Big(\mathcal{K}, u, (\mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}}); \mathsf{RW}^?_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}}\Big)$, by Theorem 8, we know
$$t \in \mathsf{lfpTx}\Big(\mathcal{K}, u, (\mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}}); \mathsf{RW}^?_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}}\Big)$$
  - If $t' \in T_{\mathsf{rd}}$, there are two cases:
    * $t' \in \left(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl})\right)$. Note that t' is a read-only transaction, which means $t \xrightarrow{\mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}}} t'$. By the property of $\mathcal{X}$ (before update) that $(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}}); \mathsf{VIS}_{\mathcal{X}} \in \mathsf{VIS}_{\mathcal{X}}$, it is known that $t \in \left(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl})\right)$, that is, $t \in \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}}$.
    * $t' \in \left(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{SO}^{-1}_{\mathcal{X}}(t^n_{cl})\right)$ and it is a read only transaction. Then we know $t \in (\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}})^{-1} \left(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{SO}^{-1}_{\mathcal{X}}(t^n_{cl})\right)$. By the property of $\mathcal{X}$ (before update) that $\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}} \in \mathsf{VIS}_{\mathcal{X}}$, it follows:

$$\begin{aligned} t &\in VIS^{-1}_{\mathcal{X}} \left(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{SO}^{-1}_{\mathcal{X}}(t^n_{cl}) \right) \\ &= \left(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl}) \right) \\ &= \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}} \end{aligned}$$

– Since `CP` satisfies `RYW` and `MR`, thus invariants I_1 and I_2 are preserved after update.

The execution test $\mathtt{ET}_{\mathtt{CP}}$ is complete with respect to the axiomatic definition

$$(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.\mathsf{AR}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}}, \lambda\mathcal{X}.\,\mathsf{SO}_{\mathcal{X}}\})$$

Assume i-*th* transaction t_i in the arbitrary order, and let view $u_i = \mathsf{getView}\big(\mathcal{X}, \mathsf{VIS}_{\mathcal{X}}^{-1}(t_i)\big)$. We pick final view as $u_i' = \mathsf{getView}\big(\mathcal{X}, (\mathsf{AR}_{\mathcal{X}}^{-1})^?(t_i) \cap \mathsf{VIS}_{\mathcal{X}}^{-1}(t_i')\big)$, if $t_i' = \min_{\mathsf{SO}}\left\{t' \;\middle|\; t_i \xrightarrow{\mathsf{SO}} t'\right\}$ is defined, otherwise $u_i' = \mathsf{getView}\big(\mathcal{X}, (\mathsf{AR}_{\mathcal{X}}^{-1})^?(t_i)\big)$. Let the $\mathcal{K} = \mathcal{K}_{\mathsf{cut}(\mathcal{X},i-1)}$. Now we prove the three parts separately.

– `MR`. By Prop. 18 since $\mathsf{VIS}_{\mathcal{X}}; \mathsf{SO}_{\mathcal{X}} \subseteq \mathsf{AR}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$ so it follows as in §F.1.
– `RYW`. For `RYW`, since $\mathsf{WR}_{\mathcal{X}}; \mathsf{SO}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{AR}_{\mathcal{X}}; \mathsf{AR}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$, the proof is as the same proof as in §F.3.
– $\mathtt{allowed}\Big((\mathsf{SO}; \mathsf{RW}_{\mathcal{K}}^?) \cup (\mathsf{WR}_{\mathcal{K}}; \mathsf{RW}_{\mathcal{K}}^?) \cup \mathsf{WW}_{\mathcal{K}}\Big)$ can be derived from Theorem 8 and

$$(\mathsf{SO}; \mathsf{RW}_{\mathcal{X}}^?) \cup (\mathsf{WR}_{\mathcal{X}}; \mathsf{RW}_{\mathcal{X}}^?) \cup \mathsf{WW}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{AR}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$$

F.8 Parallel Snapshot Isolation `PSI`

The axiomatic definition for `PSI` is

$$(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.\mathsf{VIS}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}}, \lambda\mathcal{X}.\,\mathsf{SO}_{\mathcal{X}}, \lambda\mathcal{X}.\mathsf{WW}_{\mathcal{X}}\})$$

There exist a minimum visibility such that

$$(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.(\mathsf{WR}_{\mathcal{X}} \cup \mathsf{WW}_{\mathcal{X}} \cup \mathsf{SO}); \mathsf{VIS}_{\mathcal{X}}, \lambda\mathcal{X}.\,\mathsf{SO}_{\mathcal{X}}, \lambda\mathcal{X}.\mathsf{WW}_{\mathcal{X}}\})$$

by solve the following inequalities:

$$\begin{array}{l}\mathsf{WR} \subseteq \mathsf{VIS}\\ \mathsf{WW} \subseteq \mathsf{VIS}\\ \mathsf{SO} \subseteq \mathsf{VIS}\\ \mathsf{VIS}; \mathsf{VIS} \subseteq \mathsf{VIS}\end{array}$$

It is easy to see the former implies to later. For another way round, Lemma 9.

Lemma 9. *For any abstract execution $\mathcal{X}$ under last-write-win, if it satisfies the following:*

$$(\mathsf{WR}_{\mathcal{X}} \cup \mathsf{WW}_{\mathcal{X}} \cup \mathsf{SO}_{\mathcal{X}}); \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}} \quad \mathsf{SO}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$$

There exists a new abstract execution $\mathcal{X}'$ where $T_{\mathcal{X}} = T_{\mathcal{X}'}$, $\mathsf{AR}_{\mathcal{X}} = \mathsf{AR}_{\mathcal{X}'}$, $\mathsf{VIS}_{\mathcal{X}'}; \mathsf{VIS}_{\mathcal{X}'} \subseteq \mathsf{VIS}_{\mathcal{X}'}$, and under last-write-win $\mathscr{T}_{\mathcal{X}}(t) = \mathscr{T}_{\mathcal{X}'}(t)$ for all transactions t.

Proof. we erase some visibility relation for each transaction following the order of arbitration AR until the visibility is transitive. Assume the i-*th* transaction t_i with respect to the arbitration order. Let R_i denote a new visibility for transaction t_i such that $R_i \lfloor_2 = \{t_i\}$ and the visibility relation before (including) t_i is transitive. Let $\mathcal{X}_i = \mathcal{K}_{\mathsf{cut}(\mathcal{X},i)}$ and $\mathsf{VIS}_i = \bigcup_{0 \le k \le i} R_i$. For each step, says i-*th* step, we preserve the following:

$$\mathsf{VIS}_i ; \mathsf{VIS}_i \subseteq \mathsf{VIS}_i \tag{6.10}$$

$$\forall t.\ (t, t_i) \in R_i \Rightarrow (t, t_i) \in (\mathsf{WR}_i \cup \mathsf{WW}_i \cup \mathsf{SO}_i) \tag{6.11}$$

- Base case: $i = 1$ and $R_1 = \emptyset$. Assume it is from client cl. There is no transaction committed before, so $\mathsf{VIS}_1 = \emptyset$ and $\mathsf{VIS}_1 ; \mathsf{VIS}_1 \subseteq \mathsf{VIS}_1$ as Eq. (6.10).
- Inductive case: i-*th* step. Suppose the (i-1)-*th* step satisfies Eq. (6.10) and Eq. (6.11). Let consider i-*th* step and the transaction t_i. Initially we take R_i as empty set. We first extend R_i by closing with respect to WR_i and prove that it does not affect any read from the transaction t_i. Then we will do the same for SO_i and WW_i.
  - WR_i. For any read $(\mathtt{r}, k, v) \in t_i$, there must be a transaction t_j that $t_j \xrightarrow{\mathsf{WR}(\mathcal{X}_i,k),\mathsf{AR}} t_i$ and $j < i$. We include $(t_j, t_i) \in R_i$. Let consider all the visible transactions of t_j. Assume a transaction $t' \in \mathsf{VIS}_{i-1}^{-1}(t_j)$, thus $t' \in \mathsf{VIS}_j^{-1}(t_j) = R_j^{-1}(t_j)$. It is safe to include $(t', t_i) \in R_i$ without affecting the read result, because those transaction t' is already visible for t_i in the abstract execution $\mathcal{X}$: by Eq. (6.11) we know $R_j \subseteq (\mathsf{WR}_j \cup \mathsf{SO}_j \cup \mathsf{WW}_j)^+ \subseteq (\mathsf{WR}_{\mathcal{X}} \cup \mathsf{SO}_{\mathcal{X}} \cup \mathsf{WW}_{\mathcal{X}})^+$, and by the definition of $\mathsf{WR}(\mathcal{X}_i, k)$ we know $\mathsf{WR}(\mathcal{X}_i, k) \subseteq \mathsf{VIS}_{\mathcal{X}}$.
  - Given $\mathsf{SO}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$ (and $\mathsf{WW}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$ respectively) we include (t_j, t_i) for some t_j such that $t_j \xrightarrow{\mathsf{SO}_{\mathcal{X}}} t_i$ (and $t_j \xrightarrow{\mathsf{WW}_{\mathcal{X}}} t_i$ respectively). For the similar reason as WR, it is safe to includes all the visible transactions t' for t_j, i.e. $t' \in R_j^{-1}$.

  By the construction, both Eq. (6.2) and Eq. (6.3) are preserved. Thus we have the proof.

To prove soundness, we pick an invariant for the $\mathtt{ET}_{\mathtt{PSI}}$ as the union of those for `MR` and `RYW` shown in the following:

$$I_1(\mathcal{X}, cl) = \left(\bigcup_{\left\{ t_{cl}^n \in T_{\mathcal{X}} \mid n \in \mathbb{N} \right\}} \mathsf{VIS}_{\mathcal{X}}^{-1}(t_{cl}^n) \right) \setminus T_{\mathsf{rd}}$$

$$I_2(\mathcal{X}, cl) = \left(\bigcup_{\left\{ t_{cl}^n \in T_{\mathcal{X}} \mid n \in \mathbb{N} \right\}} (\mathsf{SO}_{\mathcal{X}}^{-1})^?(t_{cl}^n) \right) \setminus T_{\mathsf{rd}}$$

where T_{rd} is all the read-only transactions included in both

$$T_{\mathsf{rd}} \in \left(\bigcup_{\left\{ t_{cl}^n \in T_{\mathcal{X}} \mid n \in \mathbb{N} \right\}} \mathsf{VIS}_{\mathcal{X}}^{-1}(t_{cl}^n) \right)$$

and

$$T_{\mathsf{rd}} \in \left(\bigcup_{\left\{ t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N} \right\}} (\mathsf{SO}^{-1}_{\mathcal{X}})^?(t^n_{cl}) \right)$$

Assume a kv-store $\mathcal{K}$, an initial and a final view u, u' a fingerprint $\mathcal{F}$ such that $\mathtt{ET}_{\mathtt{PSI}} \vdash (\mathcal{K}, u) \triangleright \mathcal{F} : (\mathcal{K}', u')$. Also choose an arbitrary cl, a transaction identifier $t^n_{cl} \in \mathsf{nextTid}(\mathcal{K}, cl)$, and an abstract execution $\mathcal{X}$ such that $\mathcal{K}_{\mathcal{X}} = \mathcal{K}$ and $I_1(\mathcal{X}, cl) \cup I_2(\mathcal{X}, cl) \subseteq \mathsf{visTx}(\mathcal{K}, u)$. We are about to prove there exists an extra set of read-only transactions T'_{rd} such that the new abstract execution $\mathcal{X}' = \mathsf{extend}(\mathcal{X}, t^n_{cl}, \mathcal{F}, \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}} \cup T'_{\mathsf{rd}})$ and:

$$\forall t.\ (t, t^n_{cl}) \in \mathsf{SO}_{\mathcal{X}'} \Rightarrow t \in \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}} \cup T'_{\mathsf{rd}} \tag{6.12}$$

$$\forall t.\ (t, t^n_{cl}) \in \mathsf{WW}_{\mathcal{X}'} \Rightarrow t \in \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}} \cup T'_{\mathsf{rd}} \tag{6.13}$$

$$\forall t.\ (t, t^n_{cl}) \in (\mathsf{SO}_{\mathcal{X}'} \cup \mathsf{WR}_{\mathcal{X}'} \cup \mathsf{WW}_{\mathcal{X}'}); \mathsf{VIS}_{\mathcal{X}'} \Rightarrow t \in \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}} \cup T'_{\mathsf{rd}} \tag{6.14}$$

$$I_1(\mathcal{X}', cl) \cup I_2(\mathcal{X}', cl) \subseteq \mathsf{visTx}(\mathcal{K}_{\mathcal{X}'}, u') \tag{6.15}$$

- The invariant I_2 implies Eq. (6.12) as the same as `RYW` in §F.3.
- Since `PSI` also satisfies `UA`, the Eq. (6.17) can be proven as the same as `UA` in §F.6.
- Eq. (6.14). Note that

  $$(t, t^n_{cl}) \in (\mathsf{SO}_{\mathcal{X}'} \cup \mathsf{WR}_{\mathcal{X}'} \cup \mathsf{WW}_{\mathcal{X}'}); \mathsf{VIS}_{\mathcal{X}'} \Rightarrow (t, t^n_{cl}) \in (\mathsf{SO}_{\mathcal{X}} \cup \mathsf{WR}_{\mathcal{X}} \cup \mathsf{WW}_{\mathcal{X}}); \mathsf{VIS}_{\mathcal{X}'}$$

  Also, recall that $\mathsf{SO}_{\mathcal{X}} = \mathsf{SO}_{\mathcal{K}}$, $\mathsf{WR}_{\mathcal{X}} = \mathsf{WR}_{\mathcal{K}}$ and $\mathsf{WW}_{\mathcal{X}} = \mathsf{WW}_{\mathcal{K}}$. Let

  $$T'_{\mathsf{rd}} = \mathsf{lfpTx}(\mathcal{K}, u, \mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}})$$

  This means

  $$\mathcal{X}' = \mathsf{extend}(\mathcal{X}, t^n_{cl}, \mathcal{F}, \mathsf{lfpTx}(\mathcal{K}, u, \mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}}) \cup T_{\mathsf{rd}})$$

  Let assume $t \xrightarrow{\mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}}} t'$ and $t' \in \mathsf{lfpTx}(\mathcal{K}, u, \mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}}) \cup T_{\mathsf{rd}}$. We have two possible cases:
  - If $t' \in \mathsf{lfpTx}(\mathcal{K}, u, \mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}})$, by Theorem 8, we know

    $$t \in \mathsf{lfpTx}(\mathcal{K}, u, \mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}})$$

  - If $t' \in T_{\mathsf{rd}}$, there are two cases:
    * $t' \in \left(\bigcup_{\{ t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N} \}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl}) \right)$. Since t' is a read-only transaction, it means $t \xrightarrow{\mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}}} t'$. By the property of $\mathcal{X}$ (before update) that $(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}}); \mathsf{VIS}_{\mathcal{X}} \in \mathsf{VIS}_{\mathcal{X}}$, it is known that $t \in \left(\bigcup_{\{ t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N} \}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl}) \right)$, that is, $t \in \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}}$.

∗ $t' \in \left(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{SO}^{-1}_{\mathcal{X}}(t^n_{cl})\right)$. Given that t' is a read only transaction, we know $t \in (\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}})^{-1} \left(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{SO}^{-1}_{\mathcal{X}}(t^n_{cl})\right)$. By the property of $\mathcal{X}$ (before update) that $\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}} \in \mathsf{VIS}_{\mathcal{X}}$, it follows:

$$\begin{aligned} t \in VIS^{-1}_{\mathcal{X}} & \left(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{SO}^{-1}_{\mathcal{X}}(t^n_{cl}) \right) \\ = & \left(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl}) \right) \\ = & \, \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}} \end{aligned}$$

– Finally the new abstract execution preserves the invariant I_1 and I_2 because `CC` satisfies `MW` and `RYW`.

The execution test $\mathtt{ET}_{\mathtt{PSI}}$ is complete with respect to the axiomatic definition

$$(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.\mathsf{VIS}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}}, \lambda\mathcal{X}.\,\mathsf{SO}_{\mathcal{X}}, \lambda\mathcal{X}.\,\mathsf{WW}_{\mathcal{X}}\})$$

Assume i-*th* transaction t_i in the arbitrary order, and let view $u_i = \mathsf{getView}\big(\mathcal{X}, \mathsf{VIS}^{-1}_{\mathcal{X}}(t_i)\big)$. We pick final view as $u'_i = \mathsf{getView}\big(\mathcal{X}, (\mathsf{AR}^{-1}_{\mathcal{X}})^?(t_i) \cap \mathsf{VIS}^{-1}_{\mathcal{X}}(t'_i)\big)$, if $t'_i = \min_{\mathsf{SO}} \left\{ t' \;\middle|\; t_i \xrightarrow{\mathsf{SO}} t' \right\}$ is defined, otherwise $u'_i = \mathsf{getView}\big(\mathcal{X}, (\mathsf{AR}^{-1}_{\mathcal{X}})^?(t_i)\big)$. Let the $\mathcal{K} = \mathcal{K}_{\mathsf{cut}(\mathcal{X}, i-1)}$. Now we prove the three parts separately.

– `MR`. By Prop. 18 since $\mathsf{VIS}_{\mathcal{X}}; \mathsf{SO}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$ so it follows as in §F.1.
– `RYW`. For `RYW`, since $\mathsf{WR}_{\mathcal{X}}; \mathsf{SO}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$, the proof is as the same proof as in §F.3.
– `UA`. Since $\mathsf{WW}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$, the proof is as the same proof as in §F.6.
– `allowed`$(\mathsf{WR}_{\mathcal{K}}\mathsf{WW}_{\mathcal{K}} \cup \cup\mathsf{SO})$. It is derived from Theorem 8 and

$$(\mathsf{WR}_{\mathcal{X}} \cup \mathsf{WW}_{\mathcal{X}} \cup \mathsf{SO}_{\mathcal{X}}); \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$$

F.9 Snapshot Isolation `SI`

The axiomatic definition for `SI` is

$$(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.\mathsf{AR}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}}, \lambda\mathcal{X}.\,\mathsf{SO}_{\mathcal{X}}, \lambda\mathcal{X}.\,\mathsf{WW}_{\mathcal{X}}\})$$

By a lemma proven in [16], for any $\mathcal{X}$ satisfies the `SI` there exists an equivalent $\mathcal{X}'$ with minimum visibility $\mathsf{VIS}_{\mathcal{X}'} \subseteq \mathsf{VIS}_{\mathcal{X}}$ satisfying

$$\left(\mathsf{RP}_{\mathsf{LWW}}, \left\{\lambda\mathcal{X}.\left((\mathsf{SO}_{\mathcal{X}'} \cup \mathsf{WW}_{\mathcal{X}'} \cup \mathsf{WR}_{\mathcal{X}'}); \mathsf{RW}^?_{\mathcal{X}'}\right); \mathsf{VIS}'_{\mathcal{X}'}, \lambda\mathcal{X}.\,(\mathsf{WW}_{\mathcal{X}'} \cup \mathsf{SO}_{\mathcal{X}'})\right\}\right)$$

Under the minimum visibility VIS all the transactions still have the same behaviour as before, meaning they do not violate last-write-win.

To prove the soundness, we pick the invariant as the following:

$$I_1(\mathcal{X}, cl) = \left(\bigcup_{\left\{t^i_{cl} \in T_{\mathcal{X}} \mid i \in \mathbb{N}\right\}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^i_{cl}) \right) \setminus T_{\mathsf{rd}}$$

$$I_2(\mathcal{X}, cl) = \left(\bigcup_{\left\{t^i_{cl} \in T_{\mathcal{X}} \mid i \in \mathbb{N}\right\}} (\mathsf{SO}^{-1}_{\mathcal{X}})^?(t^i_{cl}) \right) \setminus T_{\mathsf{rd}}$$

where T_{rd} is all the read-only transactions included in both

$$T_{\mathsf{rd}} \in \left(\bigcup_{\left\{t^i_{cl} \in T_{\mathcal{X}} \mid i \in \mathbb{N}\right\}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^i_{cl}) \right)$$

and

$$T_{\mathsf{rd}} \in \left(\bigcup_{\left\{t^i_{cl} \in T_{\mathcal{X}} \mid i \in \mathbb{N}\right\}} (\mathsf{SO}^{-1}_{\mathcal{X}})^?(t^i_{cl}) \right)$$

Assume a kv-store $\mathcal{K}$, an initial and a final view u, u' a fingerprint $\mathcal{F}$ such that $\mathtt{ET}_{\mathtt{SI}} \vdash (\mathcal{K}, u) \triangleright \mathcal{F} : (\mathcal{K}', u')$. Also choose an arbitrary cl, a transaction identifier $t^n_{cl} \in \mathsf{nextTid}(\mathcal{K}, cl)$, and an abstract execution $\mathcal{X}$ such that $\mathcal{K}_{\mathcal{X}} = \mathcal{K}$ and $I_1(\mathcal{X}, cl) \cup I_2(\mathcal{X}, cl) \subseteq \mathsf{visTx}(\mathcal{K}, u)$. We are about to prove there exists an extra set of read-only transaction T'_{rd} such that the new abstract execution $\mathcal{X}' = \mathsf{extend}(\mathcal{X}, t^n_{cl}, \mathcal{F}, \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}} \cup T'_{\mathsf{rd}})$ and

$$\forall t.\ (t, t^n_{cl}) \in \mathsf{SO}_{\mathcal{X}'} \Rightarrow t \in \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}} \cup T'_{\mathsf{rd}} \tag{6.16}$$

$$\forall t.\ (t, t^n_{cl}) \in \mathsf{WW}_{\mathcal{X}'} \Rightarrow t \in \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}} \cup T'_{\mathsf{rd}} \tag{6.17}$$

$$\forall t.\ (t, t^n_{cl}) \in \left((\mathsf{SO}_{\mathcal{X}'} \cup \mathsf{WW}_{\mathcal{X}'} \cup \mathsf{WR}_{\mathcal{X}'}); \mathsf{RW}^?_{\mathcal{X}'}\right); \mathsf{VIS}_{\mathcal{X}'} \Rightarrow t \in \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}} \cup T'_{\mathsf{rd}} \tag{6.18}$$

$$I_1(\mathcal{X}', cl) \cup I_2(\mathcal{X}', cl) \subseteq \mathsf{visTx}(\mathcal{K}_{\mathcal{X}'}, u') \tag{6.19}$$

- The invariant I_2 implies Eq. (6.16) as the same as `RYW` in §F.3.
- Since `SI` also satisfies `UA`, the Eq. (6.17) can be proven as the same as `UA` in §F.6.
- Eq. (6.18). Note that

$$\begin{aligned}(t, t^n_{cl}) \in \left((\mathsf{SO}_{\mathcal{X}'} \cup \mathsf{WW}_{\mathcal{X}'} \cup \mathsf{WR}_{\mathcal{X}'}); \mathsf{RW}^?_{\mathcal{X}'}\right); \mathsf{VIS}_{\mathcal{X}'} \\ \Rightarrow (t, t^n_{cl}) \in \left((\mathsf{SO}_{\mathcal{X}} \cup \mathsf{WW}_{\mathcal{X}} \cup \mathsf{WR}_{\mathcal{X}}); \mathsf{RW}^?_{\mathcal{X}}\right); \mathsf{VIS}_{\mathcal{X}'}\end{aligned}$$

  Also, recall that $R_{\mathcal{X}} = R_{\mathcal{K}}$ for $R \in \{\mathsf{SO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}\}$. Let

$$T'_{\mathsf{rd}} = \mathsf{lfpTx}\Big(\mathcal{K}, u, \left((\mathsf{SO}_{\mathcal{X}'} \cup \mathsf{WW}_{\mathcal{X}'} \cup \mathsf{WR}_{\mathcal{X}'}); \mathsf{RW}^?_{\mathcal{X}'}\right)\Big)$$

This means that

$$\mathcal{X}' = \mathsf{extend}\Big(\mathcal{X}, t^n_{cl}, \mathcal{F}, \mathsf{lfpTx}\Big(\mathcal{K}, u, \Big((\mathsf{SO}_{\mathcal{X}'} \cup \mathsf{WW}_{\mathcal{X}'} \cup \mathsf{WR}_{\mathcal{X}'}); \mathsf{RW}^?_{\mathcal{X}'}\Big)\Big) \cup T_{\mathsf{rd}}\Big)$$

Let assume $t \xrightarrow{\left((\mathsf{SO}_{\mathcal{X}'} \cup \mathsf{WW}_{\mathcal{X}'} \cup \mathsf{WR}_{\mathcal{X}'}); \mathsf{RW}^?_{\mathcal{X}'}\right)} t'$ and

$$t' \in \mathsf{lfpTx}\Big(\mathcal{K}, u, \Big((\mathsf{SO}_{\mathcal{X}'} \cup \mathsf{WW}_{\mathcal{X}'} \cup \mathsf{WR}_{\mathcal{X}'}); \mathsf{RW}^?_{\mathcal{X}'}\Big)\Big) \cup T_{\mathsf{rd}}$$

We have two possible cases:

- If $t' \in \mathsf{lfpTx}\Big(\mathcal{K}, u, \Big((\mathsf{SO}_{\mathcal{X}'} \cup \mathsf{WW}_{\mathcal{X}'} \cup \mathsf{WR}_{\mathcal{X}'}); \mathsf{RW}^?_{\mathcal{X}'}\Big)\Big)$, by Theorem 8, we know

$$t \in \mathsf{lfpTx}\Big(\mathcal{K}, u, \Big((\mathsf{SO}_{\mathcal{X}'} \cup \mathsf{WW}_{\mathcal{X}'} \cup \mathsf{WR}_{\mathcal{X}'}); \mathsf{RW}^?_{\mathcal{X}'}\Big)\Big)$$

- If $t' \in T_{\mathsf{rd}}$, there are two cases:
  - $t' \in \Big(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl})\Big)$. Since t' is a read-only transaction, it means $t \xrightarrow{\mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}}} t'$. By the property of $\mathcal{X}$ (before update) that $(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}}); \mathsf{VIS}_{\mathcal{X}} \in \mathsf{VIS}_{\mathcal{X}}$, it is known that $t \in \Big(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl})\Big)$, that is, $t \in \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}}$.
  - $t' \in \Big(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{SO}^{-1}_{\mathcal{X}}(t^n_{cl})\Big)$. Given that t' is a read only transaction, we know $t \in (\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}})^{-1} \Big(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{SO}^{-1}_{\mathcal{X}}(t^n_{cl})\Big)$. By the property of $\mathcal{X}$ (before update) that $\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}} \in \mathsf{VIS}_{\mathcal{X}}$, it follows:

$$\begin{aligned} t \in VIS^{-1}_{\mathcal{X}} & \left(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{SO}^{-1}_{\mathcal{X}}(t^n_{cl}) \right) \\ &= \left(\bigcup_{\{t^n_{cl} \in T_{\mathcal{X}} \mid n \in \mathbb{N}\}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl}) \right) \\ &= \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}} \end{aligned}$$

– Since `SI` satisfies `RYW` and `MR`, thus invariants I_1 and I_2 are preserved, that is, Eq. (6.19).

The execution test $\mathtt{ET}_{\mathtt{SI}}$ is complete with respect to the axiomatic definition

$$(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.\mathsf{AR}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}}, \lambda\mathcal{X}.\,\mathsf{SO}_{\mathcal{X}}, \lambda\mathcal{X}.\,\mathsf{WW}_{\mathcal{X}}\})$$

Assume i-*th* transaction t_i in the arbitrary order, and let view $u_i = \mathsf{getView}\big(\mathcal{X}, \mathsf{VIS}^{-1}_{\mathcal{X}}(t_i)\big)$. We pick final view as $u'_i = \mathsf{getView}\big(\mathcal{X}, (\mathsf{AR}^{-1}_{\mathcal{X}})^?(t_i) \cap \mathsf{VIS}^{-1}_{\mathcal{X}}(t'_i)\big)$, if $t'_i = \min_{\mathsf{SO}} \Big\{t' \;\Big|\; t_i \xrightarrow{\mathsf{SO}} t'\Big\}$ is defined, otherwise $u'_i = \mathsf{getView}\big(\mathcal{X}, (\mathsf{AR}^{-1}_{\mathcal{X}})^?(t_i)\big)$. Let the $\mathcal{K} = \mathcal{K}_{\mathsf{cut}(\mathcal{X}, i-1)}$. Now we prove the three parts separately.

- $\texttt{MR}$. By Prop. 18 since $\mathsf{VIS}_{\mathcal{X}}; \mathsf{SO}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$ so it follows as in §F.1.
- $\texttt{RYW}$. For $\texttt{RYW}$, since $\mathsf{WR}_{\mathcal{X}}; \mathsf{SO}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$, the proof is as the same proof as in §F.3.
- $\texttt{UA}$. Since $\mathsf{WW}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$, the proof is as the same proof as in §F.6.
- $\texttt{allowed}\Big(\Big((\mathsf{SO}_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}}); \mathsf{RW}^{?}_{\mathcal{K}}\Big)\Big)$. It is derived from Theorem 8 and

$$\Big((\mathsf{SO}_{\mathcal{X}} \cup \mathsf{WW}_{\mathcal{X}} \cup \mathsf{WR}_{\mathcal{X}}); \mathsf{RW}^{?}_{\mathcal{X}}\Big); \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{AR}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$$

F.10 Serialisability $\texttt{SER}$

The execution test $\texttt{ET}_{\texttt{SER}}$ is sound with respect to the axiomatic definition

$$(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.\mathsf{AR}\})$$

We pick the invariant as $I(\mathcal{X}, cl) = \emptyset$, given the fact of no constraint on the view after update. Assume a kv-store $\mathcal{K}$, an initial and a final view u, u' a fingerprint $\mathcal{F}$ such that $\texttt{ET}_{\texttt{SER}} \vdash (\mathcal{K}, u) \triangleright \mathcal{F} : (\mathcal{K}', u')$. Also choose an arbitrary cl, a transaction identifier $t \in \mathsf{nextTid}(\mathcal{K}, cl)$, and an abstract execution $\mathcal{X}$ such that $\mathcal{K}_{\mathcal{X}} = \mathcal{K}$ and $I(\mathcal{X}, cl) = \emptyset \subseteq \mathsf{visTx}(\mathcal{K}, u)$. Let $\mathcal{X}' = \mathsf{extend}(\mathcal{X}, t, \mathsf{visTx}(\mathcal{K}, u), \mathcal{F})$. Note that since the invariant is empty set, it remains to prove there exists a set of read-only transactions T_{rd} such that:

$$\forall t'.\ t' \xrightarrow{\mathsf{AR}_{\mathcal{X}'}} t \Rightarrow t' \in \mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}}$$

Since the abstract execution satisfies the constraint for $\texttt{SER}$, i.e. $\mathsf{AR} \subseteq \mathsf{VIS}$, we know $\mathsf{AR} = \mathsf{VIS}$. Since $\mathsf{visTx}(\mathcal{K}, u)$ contains all transactions that write at least a key, we can pick a T_{rd} such that $\mathsf{visTx}(\mathcal{K}, u) \cup T_{\mathsf{rd}} = T_{\mathcal{X}}$, which gives us the proof.

The execution test $\texttt{ET}_{\texttt{UA}}$ is complete with respect to the axiomatic definition $(\mathsf{RP}_{\mathsf{LWW}}, \{\lambda\mathcal{X}.\mathsf{AR}_{\mathcal{X}}\})$. Assume i-$th$ transaction t_i in the arbitrary order, and let view $u_i = \mathsf{getView}\big(\mathcal{X}, \mathsf{VIS}^{-1}_{\mathcal{X}}(t_i)\big)$. We also pick any final view such that $u'_i \subseteq \mathsf{getView}\big(\mathcal{X}, (\mathsf{AR}^{-1}_{\mathcal{X}})^{?}(t_i)\big)$. Note that there is nothing to prove for u'_i, Now we need to prove the following:

$$\forall k, j.\ 0 \leq j < |\mathcal{K}_{\mathsf{cut}(\mathcal{X}, i-1)}(k)| \Rightarrow j \in u_i(k)$$

Because $\mathsf{VIS}^{-1}(t_i) = \mathsf{AR}^{-1}(t_i) = \big\{t \bigm| t \in \mathcal{K}_{\mathsf{cut}(\mathcal{X}, i-1)}\big\}$, so for any key k and index j such that $0 \leq j < |\mathcal{K}_{\mathsf{cut}(\mathcal{X}, i-1)}(k)|$, the j-$th$ version of the key contains in the view, i.e. $j \in u\,(k)$.

G Program Analysis

We give applications of our theory aimed at showing the robustness of a transactional library against a given consistency model. The first application considers a single counter library, and proves that it is robust against Parallel Snapshot Isolation. We present a general robustness conditions for `WSI` and then show multiple-counter example and a banking example [3] are robust against `WSI`.

G.1 Single counter

We start by reviewing the transactional code for the increment and read operations provided by a counter object over a key k, denoted as $\mathsf{inc}(k)$ and $\mathsf{read}(k)$, respectively.

$$\mathsf{inc}(k) = \begin{bmatrix} \mathtt{a} := [k]\,; \\ [k] := \mathtt{a} + 1; \end{bmatrix} \qquad\qquad \mathsf{read}(k) = [\mathtt{a} := [k]\,;]$$

Clients can interact with the key-value store only by invoking the $\mathsf{inc}(k)$ and $\mathsf{read}(k)$ operations. A transactional library is a set of transactional operations. For a single counter over key k, we define the transactional library $\mathsf{Counter}(k) = \{\mathsf{inc}(k), \mathsf{read}(k)\}$, while for multiple counters over a set of keys $\mathsf{K} = \{k_i\}_{i\in I}$, respectively, we define $\mathsf{Counter}(\mathsf{K}) = \bigcup_{i\in I} \mathsf{Counter}(k_i)$.

KV-store semantics of a transactional library Given the transactional code $\left[\mathtt{T}\right]$, we define $\mathcal{F}(\mathcal{K}, u, \left[\mathtt{T}\right])$ to be the fingerprint that would be produced by a client that has view u over the kv-store $\mathcal{K}$, upon executing $\left[\mathtt{T}\right]$. For the $\mathsf{inc}(k)$ and $\mathsf{read}(k)$ operations discussed above, we have that

$$\mathcal{F}(\mathcal{K}, u, \mathsf{inc}(k)) = \{(\mathtt{r}, k, n), (\mathtt{w}, k, n+1) \mid n = \mathsf{snapshot}(\mathcal{K}, u)(k)\}$$

and

$$\mathcal{F}(\mathcal{K}, u, \mathtt{read}(k)) = \{(\mathtt{r}, k, n) \mid n = \mathsf{snapshot}(\mathcal{K}, u)(k)\}$$

Given an execution test $\mathtt{ET}$, and a transactional library $L = \left\{\left[\mathtt{T}_i\right]\right\}_{i\in I}$, we define the set of valid $\mathtt{ET}$-traces for L as the set $\mathtt{Traces}(\mathtt{ET}, \left\{\left[\mathtt{T}_i\right]\right\}_{i\in I})$ of $\mathtt{ET}$-traces in which only $\mathtt{ET}$-reductions of the form

$$(\mathcal{K}_0, \mathcal{U}_0) \xrightarrow{(cl_0, \lambda_0)}_{\mathtt{ET}} (\mathcal{K}_1, \mathcal{U}_1) \xrightarrow{(cl_1, \lambda_1)}_{\mathtt{ET}} \cdots \xrightarrow{(cl_{n-1}, \lambda_{n-1})}_{\mathtt{ET}} (\mathcal{K}_n, \mathcal{U}_n),$$

where for any $j = 0, \cdots, n-1$, either $\lambda_j = \varepsilon$ or $\lambda_j = \mathcal{F}(\mathcal{K}_j, \mathcal{U}_j(cl_j), \left[\mathtt{T}_i\right])$ for some $i \in I$. Henceforth we commit an abuse of notation and write $(\mathcal{K}, \mathcal{U}) \xrightarrow{(cl, \left[\mathtt{T}\right])}_{\mathtt{ET}} (\mathcal{K}', \mathcal{U}')$ in lieu of $(\mathcal{K}, \mathcal{U}) \xrightarrow{(cl, \mathcal{F}(\mathcal{K}, \mathcal{U}(cl), \left[\mathtt{T}\right])}_{\mathtt{ET}} (\mathcal{K}', \mathcal{U}')$. We also let $\mathsf{KVStores}(\mathtt{ET}, \left\{\left[\mathtt{T}_i\right]\right\}_{i\in I})$ be the set of kv-stores that can be obtained when clients can only perform operations from $\left\{\left[\mathtt{T}_i\right]\right\}_{i\in I}$ under the execution test $\mathtt{ET}$. Specifically,

$$\mathsf{KVStores}(\mathtt{ET}, \left\{\left[\mathtt{T}_i\right]\right\}_{i\in I}) \triangleq \left\{\mathcal{K} \;\middle|\; \left((\mathcal{K}_0, \mathcal{U}_0) \xrightarrow{\cdot}_{\mathtt{ET}} \cdots \xrightarrow{\cdot}_{\mathtt{ET}} (\mathcal{K}, \_)\right) \in \mathsf{Traces}(\mathtt{ET}, \left\{\left[\mathtt{T}_i\right]\right\}_{i\in I})\right\}$$

Anomaly of a single counter under Causal Consistency It is well known that the transactional library consisting of a single counter over a single key, $\mathsf{Counter}(k)$, implemented on top of a kv-store guaranteeing Causal Consistency, leads to executions over the kv-store that cannot be simulated by the same transactional library implemented on top of a serialisable kv-store. For simplicity, let us assume that $\textsc{Keys} = \{k\}$. Let $\mathcal{K}_0 = [k \mapsto (0, t_0, \emptyset)]$, $\mathcal{K}_1 = [k \mapsto (0, t_0, \{t^1_{cl_1}\}::(0, t^1_{cl_1}, \emptyset)$, $\mathcal{K}_2 = [k \mapsto (0, t_0, \{t^1_{cl_1}, t^1_{cl_2}\})::(0, t^1_{cl_1}, \emptyset)::(0, t^1_{cl_2}, \emptyset)$. Let also $u_0 = [k \mapsto 0]$. Then we have that

$$(\mathcal{K}_0, [cl_1 \mapsto u_0, cl_2 \mapsto u_0]) \xrightarrow{(cl_1, \mathsf{inc(k)})}_{\mathtt{ET_{CC}}} (\mathcal{K}_1, [cl_1 \mapsto \_, cl_2 \mapsto u_0]) \xrightarrow{(cl_1, \mathsf{inc(k)})}_{\mathtt{ET_{CC}}} (\mathcal{K}_2, \_).$$

By looking at the kv-store $\mathcal{K}_2$, we immediately find a cycle in the graph induced by the relations $\mathsf{SO}_{\mathcal{K}_2}, \mathsf{WR}_{\mathcal{K}_2}, \mathsf{WW}_{\mathcal{K}_2}, \mathsf{RW}_{\mathcal{K}_2}$: $t^1_{cl_1} \xrightarrow{\mathsf{RW}} t^1_{cl_2} \xrightarrow{\mathsf{RW}} t^1_{cl_1}$. Following from Theorem 1, then which proves that $\mathcal{K}_2$ is not included in $\mathsf{CM}(\mathtt{ET_{SER}})$, i.e. it is not serialisable.

Robustness of a Single counter under Parallel Snapshot Isolation Here we show that the single counter library $\mathsf{Counter}(k)$ is robust under any consistency model that guarantees both write conflict detection (formalised by the execution test $\mathtt{ET_{UA}}$), monotonic reads (formalised by the execution test $\mathtt{ET_{MR}}$) and read your writes (formalised by the execution test $\mathtt{ET_{RYW}}$). Because $\mathtt{ET_{PSI}}$ guarantees all such consistency guarantees, i.e. $\mathsf{CM}(\mathtt{ET_{PSI}}) \subseteq \mathsf{CM}(\mathtt{ET_{MR}} \cap \mathtt{ET_{RYW}} \cap \mathtt{ET_{UA}})$, then it also follows that a single counter is robust under Parallel Snapshot Isolation.

Proposition 19. *Let $\mathcal{K} \in \mathsf{KVStores}(\mathtt{ET_{UA}} \cap \mathtt{ET_{MR}} \cap \mathtt{ET_{RYW}}, \mathsf{Counter}(k))$. Then there exist $\{t_i\}_{i=1}^n$ and $\{T_i\}_{i=0}^n$ such that*

$$\mathcal{K}(k) = ((0, t_0, T_0 \uplus \{t_1\})::\cdots::(n-1, t_{n-1}, T_{n-1} \uplus \{t_n\}))::(n, t_n, T_n) \quad (7.1)$$

$$\forall i : 0 \leq i \leq n.\ T_i \cap \{t_i\}_{i=0}^n = \emptyset \quad (7.2)$$

$$\forall t, t', i, j : 0 \leq i, j \leq n.\ t \xrightarrow{\mathsf{SO}} t' \wedge t \in \{t_i\} \cup T_i \Rightarrow \begin{pmatrix} (t' = t_j \Rightarrow i < j) \wedge \\ (t' \in T_j \Rightarrow i \leq j) \end{pmatrix} \quad (7.3)$$

Proof. It suffices to prove that the properties (7.1),(7.2), (7.3) given in Prop. 19, are invariant under $(\mathtt{ET_{MR}} \cap \mathtt{ET_{RYW}} \cap \mathtt{ET_{UA}})$-reductions of the form

$$(\mathcal{K}, \mathcal{U}) \xrightarrow{(cl, \mathsf{inc}(k))}_{\mathtt{ET_{UA}} \cap \mathtt{ET_{MR}} \cap \mathtt{ET_{RYW}}} (\mathcal{K}', \mathcal{U}') \quad (7.4)$$

$$(\mathcal{K}, \mathcal{U}) \xrightarrow{(cl, \mathsf{read}(k))}_{\mathtt{ET_{UA}} \cap \mathtt{ET_{MR}} \cap \mathtt{ET_{RYW}}} (\mathcal{K}', \mathcal{U}). \quad (7.5)$$

To this end, we will need the following auxiliary result which holds for any configuration $(\mathcal{K}, \mathcal{U})$ that can be obtained under the execution test $\mathtt{ET_{RYW}} \cap \mathtt{ET_{MR}}$:

$$\begin{aligned} &\forall i, j, n, m, cl, k.\ t^n_{cl} \in \{\mathsf{w}(\mathcal{K}(k, i))\} \cup \mathsf{rs}(\mathcal{K}(k, i)) \\ &\quad \wedge t^m_{cl} \in \{\mathsf{w}(\mathcal{K}(k, j))\} \cup \mathsf{rs}(\mathcal{K}(k, j)) \wedge m < n \wedge i \in \mathcal{U}\,(cl)\,(k) \Rightarrow j \in \mathcal{U}\,(cl)\,(k) \end{aligned} \quad (7.6)$$

Suppose that there exist two sets $\{t_i\}_{i=1}^n$ and $\{T_i\}_{i=0}^n$ such that $(\mathcal{K}, \{t_i\}_{i=1}^n, \{T_i\}_{i=0}^n)$ satisfies the properties (7.1)-(7.3). We prove that, for transitions of the form (7.4)-(7.5), there exist an index m and two collections $\{t_i\}_{i=1}^m$, $\{T'_i\}_{i=0}^m$ such that $(\mathcal{K}', \{t_i\}_{i=1}^m, \{T'_i\}_{i=0}^m)$ satisfies the properties (7.1)-(7.3). We consider the two transitions separately.

– Assume that

$$(\mathcal{K},\mathcal{U}) \xrightarrow{(cl,\mathsf{inc}(k))}_{\mathtt{ET}_{\mathtt{UA}} \cap \mathtt{ET}_{\mathtt{MR}} \cap \mathtt{ET}_{\mathtt{RYW}}} (\mathcal{K}',\mathcal{U}')$$

for some $cl, \mathcal{K}', \mathcal{U}'$. Let $n+1 = |\mathcal{K}(k)|$. Because of the definition of $\mathtt{ET}_{\mathtt{UA}}$, we must have that $\mathcal{U}(cl) = [k \mapsto \{0, \cdots, n\}]$. Also, because $\mathcal{K}$ satisfies (7.1), we have that $\mathsf{snapshot}(\mathcal{K},\mathcal{U}(cl))(k) = n$. In particular, $\mathcal{F}(k,\mathcal{U}(cl), \mathsf{inc}(k)) = \{(\mathtt{r}, k, n), (\mathtt{w}, k, n+1)\}$. Thus we have that

$$\mathcal{K}' \in \mathsf{update}(\mathcal{K},\mathcal{U}(cl), cl, \{(\mathtt{r}, k, n), (\mathtt{w}, k, n+1)\})$$

Let t_{n+1} be the transaction identifier chosen to update $\mathcal{K}$, i.e.

$$\mathcal{K}' = \mathsf{update}(\mathcal{K},\mathcal{U}(cl), t_{n+1}, \{(\mathtt{r}, k, n), (\mathtt{w}, k, n+1)\})$$

where $t_{n+1} \in \mathsf{nextTid}(\mathcal{K}, cl)$; let also $T_{n+1} = \emptyset$. Then we have the following:

- $(\mathcal{K}', \{t_i\}_{i=1}^{n+1}, \{T_i\}_{i=0}^{n+1})$ satisfies Property (7.1). Recall that $(\mathcal{K}, \{t_i\}_{i=1}^n, \{T_i\}_{i=0}^n)$ satisfies (7.1), i.e.

$$\mathcal{K}(k) = ((0, t_0, T_0 \uplus \{t_1\}) :: \cdots :: (n-1, t_{n-1}, T_{n-1} \uplus \{t_n\})) :: (n, t_n, T_n).$$

  It follows that $\mathcal{K}'(k) = ((0, t_0, T_0 \uplus \{t_1\}) :: \cdots :: (n-1, t_{n-1}, T_n \uplus \{t_{n+1}\})) :: (n+1, t_{n+1}, T_{n+1})$, where we recall that $T_{n+1} = \emptyset$.
- $(\mathcal{K}', \{t_i\}_{i=1}^{n+1}, \{T_i\}_{i=0}^{n+1})$ satisfies Property (7.2). Let $i = 0, \cdots, n+1$. If $i = n+1$, then $T_i = \emptyset$, from which $T_i \cap \{t_j\}_{j=0}^{n+1} = \emptyset$ follows. If $i < n+1$, then because $(\mathcal{K}, \{t_i\}_{i=1}^n, \{T_i\}_{i=0}^n)$ satisfies Property (7.2), then $T_i \cap \{t_j\}_{j=0}^n = \emptyset$. Finally, because t_{n+1} was chosen to be fresh with respect to the transaction identifiers appearing in $\mathcal{K}$, and $T_i \subseteq \mathsf{rs}(\mathcal{K}(k,i))$, then we also have that $T_i \cap \{t_{n+1}\} = \emptyset$.
- $(\mathcal{K}', \{t_i\}_{i=1}^{n+1}, \{T_i\}_{i=0}^{n+1})$ satisfies Property (7.3). Let t, t' be such that $t \xrightarrow{\mathsf{SO}} t'$. Choose two arbitrary indexes $i, j = 0, \cdots, n+1$, and assume that $t \in \{t_i\} \cup T_i$. Note that if $i \leq n$, $j \leq n$, then because $(\mathcal{K}, \{t_i\}_{i=1}^n, \{T_i\}_{i=0}^n)$ satisfies Property (7.3), then if $t' = t_j$ it follows that $i < j$, and if $t' \in T_j$ it follows that $i \leq j$, as we wanted to prove. If $t \in \{t_{n+1}\} \cup T_{n+1}$, then it must be $t = t_{n+1}$ because $T_{n+1} = \emptyset$. Recall that t_{n+1} is the transaction identifier that was used to update $\mathcal{K}$ to $\mathcal{K}'$, i.e. $\mathcal{K}' = \mathsf{update}(\mathcal{K},\mathcal{U}(cl), t_{n+1}, \_)$. By definition of update, it follows that $t_{n+1} \in \mathsf{nextTid}(\mathcal{K}, cl)$, and because $t_{n+1} \xrightarrow{\mathsf{SO}} t'$, then t' cannot appear in $\mathcal{K}'$. In particular, $t' \notin \{t_j\}_{j=0}^{n+1} \cup \bigcup \{T_j\}_{j=0}^{n+1}$, hence in this case there is nothing to prove. Finally, if $t' \in \{t_{n+1}\} \cup T_{n+1}$, then it must be the case that $t' = t_{n+1}$. If $t = t_j$, because $t \xrightarrow{\mathsf{SO}} t'$ and $t' = t_{n+1}$, it cannot be $t = t_{n+1}$, hence it must be $i \leq n < n+1$.

– Suppose that

$$(\mathcal{K},\mathcal{U}) \xrightarrow{(cl,\mathsf{read}(k))}_{\mathtt{ET_{UA}}\cap\mathtt{ET_{MR}}\cap\mathtt{ET_{RYW}}} (\mathcal{K}',\mathcal{U}').$$

As in the previous case, we have that $\mathcal{K}' = \mathsf{update}(\mathcal{K},\mathcal{U}\left(cl\right),t,\{(\mathtt{r},k,i)\})$, where $m = \mathsf{snapshot}(\mathcal{K},\mathcal{U}\left(cl\right))(k)$ - in particular, because $(\mathcal{K},\{t_i\}_{i=1}^n,\{T_i\}_{i=0}^n$ satisfies Property (7.1), then it must be the case that $m = \max_<(\mathcal{U}\left(cl\right)\left(k\right))$ - and $t \in \mathsf{nextTid}(\mathcal{K},cl)$. For $i = 0,\cdots,n$, let $T_i' := T_i$ if $i \neq m$, $T_i' = T_i \cup \{t\}$ if $i = m$. Then we have that $(\mathcal{K}',\{t_i\}_{i=0}^n,\{T_i'\}_{i=0}^n)$ satisfies properties (7.1)-(7.3). Putting all these facts together, we obtain the following:

- $(\mathcal{K}',\{t_i\}_{i=0}^n,\{T_i'\}_{i=0}^n)$ satisfies Property (7.1). WIthout loss of generality, suppose that $m < n$. Because $(\mathcal{K},\{t_i\}_{i=0}^n,\{T_i\}_{i=0}^n)$ satisfies Property (7.1), we have that

  $$\mathcal{K}(k) = ((0,t_0,T_0 \uplus \{t_1\})::\cdots::(m,t_m,T_m \uplus \{t_{m+1}\})::\cdots::(n-1,t_{n-1},T_{n-1} \uplus \{t_n\}))::(n,t_n,T_n),$$

  and from the definition of update it follows that

  $$\begin{aligned}\mathcal{K}(k) &= ((0,t_0,T_0 \uplus \{t_1\})::\cdots::(m,t_m,T_m \cup \{t\} \uplus \{t_{m+1}\})::\cdots::(n-1,t_{n-1},T_{n-1} \uplus \{t_n\}))::(n,t_n,T_n)\\ &= ((0,t_0,T_0' \uplus \{t_1\})::\cdots::(m,t_m,T_m' \uplus \{t_{m+1}\})::\cdots::(n-1,t_{n-1},T_{n-1} \uplus \{t_n\}))::(n,t_n,T_n)\end{aligned}$$

- $(\mathcal{K}',\{t_i\}_{i=0}^n,\{T_i'\}_{i=0}^n)$ satisfies Property (7.2). Recall that $m = \max_<(\mathcal{U}\left(cl\right)\left(k\right))$; let $i = 0,\cdots,n$.
  Let again $i = \max_<\mathcal{U}\left(cl\right)\left(k\right)$. If $i \neq m$, then $T_i' = T_i$, and because $(\mathcal{K},\{t_i\}_{i=0}^n,\{T_i\}_{i=0}^n)$ satisfies Property (7.2) we have that $T_i' \cap \{t_i\}_{i=0}^n = \emptyset$. If $i = m$, then we have that $T_i' = T_m' = T_m \cup \{t\}$, where we recall that $t \in \mathsf{nextTid}(\mathcal{K},cl)$. Because $(\mathcal{K},\{t_i\}_{i=0}^n,\{T_i\}_{i=0}^n)$ satisfies Property (7.2), we have that $T_m \cap \{t_i\}_{i=0}^n = \emptyset$. Finally, because $t \in \mathsf{nextTid}(\mathcal{K},cl)$, then it must be the case that for any $i = 0,\cdots,n$, $t \notin \{\mathsf{w}(\mathcal{K}'(k,i))\}_{i=0}^m = \{t_i\}_{i=0}^m$, where the last equality follows because we have already proved that $(\mathcal{K}',\{t_i\}_{i=0}^n,\{T_i'\}_{i=0}^n)$ satisfies Property (7.1).
- $(\mathcal{K}',\{t_i\}_{i=0}^n,\{T_i'\}_{i=0}^n)$ satisfies Property (7.3). Let t',t'' be such that $t' \xrightarrow{\mathsf{SO}} t''$. Suppose also that $t' \in \{t_i\} \cup T_i'$ for some $i = 0,\cdots,n$. We consider two different cases:
  - $t' = t_i$. Suppose then that $t'' = t_j$ for some $j = 0,\cdots,n$. Because $(\mathcal{K},\{t_i\}_{i=0}^n,\{T_i\}_{i=0}^n)$ satisfies Property (7.3), then it must be the case that $i < j$. Otherwise, suppose that $t'' \in T_j'$ for some $j = 0,\cdots,n$. If $j \neq m$, then $T_j' = T_j$, and because $(\mathcal{K},\{t_i\}_{i=0}^n,\{T_i\}_{i=0}^n)$ satisfies Property (7.3), we have that $i \leq j$. Otherwise, $T_j' = T_m' = T_m \cup \{t\}$. Without loss of generality, in this case we can assume that $t'' = t$ (we have already shown that if $t'' \in T_j$, then it must be $i \leq j$. Recall that $j = m = \max(\mathcal{U}\left(cl\right)\left(k\right))$, by the Definition of $\mathtt{ET_{UA}}$ it must be the case that $\mathcal{U}\left(cl\right) = [k \mapsto \{0,\cdots,j\}]$. It also follows that $t = t_{cl}^p$ for some $p \geq 0$, and because $t' \xrightarrow{\mathsf{SO}} t'' = t$, then $t' = t_{cl}^q$ for some $q < p$. Because of Property (7.6), and because $t' = t_i = \mathsf{w}(\mathcal{K}(k,i))$, then it must be the case the case that $i \in \mathcal{U}\left(cl\right)\left(k\right)$, hence $i \leq m = j$.

  - $t' \in T'_i$. We need to distinguish the cases $i \neq m$, leading to $T'_i = T_i$, or $i = m$, in which case $T'_i = T'_m = T_m \cup \{t\}$. If either $i \neq m$, or $i = m$ and $t \in T_m$, then we can proceed as in the case $t' = t_i$. Otherwise, suppose that $i = m$ and $t' = t$. Then, because $t' \xrightarrow{\mathsf{SO}} t''$, and $t \in \mathsf{nextTid}(\mathcal{K}, cl)$, it must be the case that $t = t^p_{cl}$ for some $p \geq 0$, and whenever $t_{cl} \in k$, then $t_{cl} \xrightarrow{\mathsf{SO}} t$. In particular we cannot have that $t'' \in k$, because $t \xrightarrow{\mathsf{SO}} t''$, which concludes the proof.
- $(\mathcal{K}', \mathcal{U}')$ satisfies Property (7.6).

Corollary 5. *Given* $\mathcal{K} \in \mathsf{KVStores}(\mathtt{ET}_{\mathtt{UA}} \cap \mathtt{ET}_{\mathtt{MR}} \cap \mathtt{ET}_{\mathtt{RYW}}, \mathsf{Counter})$*, then* $\mathsf{graphOf}(\mathcal{K})$ *is acyclic.*

Proof. Let $\{t_i\}_{i=1}^n$, $\{T_i\}_{i=0}^n$ be such that $(\{t_i\}_{i=1}^n, \{T_i\}_{i=0}^n)$ satisfies properties (7.1)-(7.3). First, we define a partial order between transactions appearing in $\mathcal{K}$ as the smallest relation $\dashrightarrow$ such that for any t, t', t'' and $i, j = 0, \cdots, n$

$$\begin{array}{ll} t \in T_i & \Rightarrow t_i \dashrightarrow t, \\ i < j & \Rightarrow t_i \dashrightarrow t_j, \\ t \in T_i \wedge i < j & \Rightarrow t \dashrightarrow t_j \\ t, t' \in T_i \wedge t \xrightarrow{\mathsf{SO}} t' & \Rightarrow t \dashrightarrow t' \\ t \dashrightarrow t' \rightarrow t'' & \Rightarrow t \dashrightarrow t'' \end{array}$$

It is immediate that if $t \dashrightarrow t'$ then either $t \in \{t_i\} \cup T_i$, $t' = \{t_j\} \cup T_j$ for some i, j such that $i < j$, or $t = t_i$, $t' \in T_i$, or $t, t' \in T_i$ and $t \xrightarrow{\mathsf{SO}_{\mathcal{K}}} t'$. A consequence of this fact, is that $\dashrightarrow$ is irreflexive.

Next, observe that we have the following:

- whenever $t \xrightarrow{\mathsf{WR}_{\mathcal{K}}} t'$, then there exists an index $i = 0, \cdots, n$ such that $t = t_i$, and either $i < n$ and $t' \in T_i \cup \{t_{i+1}\}$, or $i = n$ and $t' \in T_i$: by definition, we have that $t \dashrightarrow t'$;
- whenever $t, \xrightarrow{\mathsf{WW}_{\mathcal{K}}} t'$, then there exist two indexes $i, j : 0 \leq i < j \leq n$ such that $t = t_i, t' = t_j$; again, we have that $t \dashrightarrow t'$,
- whenever $t \xrightarrow{\mathsf{RW}_{\mathcal{K}}} t'$, then there exist two indexes $i, j : 0 \leq i < j \leq n$ such that either $t \in T_i$ and $t' = t_j$, or $t = t_{i+1}$, $i + 1 < j$ and $t' = t_j$; in both cases, we obtain that $t \dashrightarrow t'$,
- whenever $t \xrightarrow{\mathsf{SO}_{\mathcal{K}}} t'$, then $t \in \{t_i\} \cup T_i$ for some $i = 0, \cdots, n$, and either $t' = t_j$ for some $i < j$, or $t' \in T_j$ for some $i \leq j$; it follows that $t \dashrightarrow t'$.

We have proved that $\dashrightarrow$ is an irreflexive relation, and it contains $(\mathsf{SO}_{\mathcal{K}} \cup \mathsf{WR}_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{RW}_{\mathcal{K}})^+$; because any subset of an irreflexive relation is itself irreflexive, we obtain that $\mathsf{graphOf}(\mathcal{K})$ is acyclic.

Corollary 6. $\mathsf{KVStores}(\mathtt{ET}_{\mathtt{PSI}}, \mathsf{Counter}(k)) \subseteq \mathsf{KVStores}(\mathtt{ET}_{\mathtt{SER}}, \mathsf{Counter}(k))$.

Proof. Let $\mathcal{K} \in \mathsf{KVStores}(\mathtt{ET}_{\mathtt{PSI}}, \mathsf{Counter}(k))$. Because $\mathtt{ET}_{\mathtt{PSI}} \supseteq \mathtt{ET}_{\mathtt{MR}} \cap \mathtt{ET}_{\mathtt{RYW}} \cap \mathtt{ET}_{\mathtt{UA}}$, we have that $\mathcal{K} \in \mathsf{KVStores}(\mathtt{ET}_{\mathtt{MR}} \cap \mathtt{ET}_{\mathtt{RYW}} \cap \mathtt{ET}_{\mathtt{UA}}, \mathsf{Counter}(k))$. By Cor. 5 we have

that $\mathsf{graphOf}(\mathcal{K})$ is acyclic. We can now employ the construction outlined in [18] to recover an abstract execution $\mathcal{X} = (T_{\mathcal{K}}, \mathsf{VIS}, \mathsf{AR})$ such that $\mathsf{SO} \subseteq \mathsf{VIS}$ and $\mathsf{AR} \subseteq \mathsf{VIS}$, and $\mathsf{graphOf}(\mathcal{X}) = \mathsf{graphOf}(\mathcal{K})$. Finally, the results from §F.10 establish that, from $\mathcal{X}$ we can recover a $\mathtt{ET}_{\mathtt{SER}}$-trace in $\mathsf{Traces}(\mathtt{ET}_{\mathtt{SER}}, \mathsf{Counter}(k))$ whose last configuration is $(\mathcal{K}', \_)$, and $\mathsf{graphOf}(\mathcal{K}') = \mathsf{graphOf}(\mathcal{X}) = \mathsf{graphOf}(\mathcal{K})$, leading to $\mathcal{K}' = \mathcal{K}$. It follows that $\mathcal{K} \in \mathsf{KVStores}(\mathtt{ET}_{\mathtt{SER}}, \mathsf{Counter}(k))$.

G.2 Robust against WSI

Definition 33. *A key-value store $\mathcal{K}$ is* WSI *safe if $\mathcal{K}$ is reachable from executing an program* P *from an initial configuration Γ_0,* i.e. $\mathtt{ET}_{\mathtt{WSI}} \vdash \Gamma_0, \mathtt{P} \rightarrow (\mathcal{K}, \mathcal{U}), \mathtt{P}'$, *and $\mathcal{K}$ satisfies the following:*

$$\forall t, k, k', i, j.\ (t \in \mathsf{rs}(\mathcal{K}(k,i)) \Rightarrow t \neq \mathsf{w}(\mathcal{K}(k,i))) \Rightarrow t \neq \mathsf{w}(\mathcal{K}(k',j)) \tag{7.7}$$

$$\forall t, k, i.\ t \neq t_0 \exists j.\ t = \mathsf{w}(\mathcal{K}(k,i)) \Rightarrow t \in \mathsf{rs}(\mathcal{K}(k,j)) \tag{7.8}$$

$$\forall t, k, k', i, j.\ t \neq t_0 \exists k.\ t = \mathsf{w}(\mathcal{K}(k,i)) \wedge t \in \mathsf{rs}(\mathcal{K}(k',j)) \Rightarrow t = \mathsf{w}(\mathcal{K}(k',j)) \tag{7.9}$$

Theorem 9. *If a key-value store $\mathcal{K}$ is* WSI *safe, it is robust against* WSI.

Proof. Assume a kv-store $\mathcal{K}$ that is WSI safe. Given Def. 33 that $\mathcal{K}$ is reachable under WSI therefore CC and UA since $\mathtt{CC} \cup \mathtt{UA} \subseteq \mathtt{WSI}$, it is easy to derive the following properties:

$$\forall t, t'.\ t \xrightarrow{\mathsf{WR} \cup \mathsf{SO} \cup \mathsf{WW}} t' \Rightarrow t \neq t' \tag{7.10}$$

$$\forall k, i, j.\ t = \mathsf{w}(\mathcal{K}(k,i)) \wedge t \in \mathsf{rs}(\mathcal{K}(k,j)) \Rightarrow i = j + 1 \tag{7.11}$$

Eq. (7.10) To prove the robustness, it is sufficient to prove that the relation $(\mathsf{WW} \cup \mathsf{WR} \cup \mathsf{RW} \cup \mathsf{SO})^+$ is irreflexive, that is, for any transactions t and t':

$$t \xrightarrow{(\mathsf{WW} \cup \mathsf{WR} \cup \mathsf{RW} \cup \mathsf{SO})^+} t' \Rightarrow t \neq t'$$

We prove that by contradiction. Let assume $t = t'$. By Eq. (7.10), it must be the case that the cycle contains RW, which means there exists t_1 to t_n such that

$$t = t_1 \xrightarrow{R^*} t_2 \xrightarrow{\mathsf{RW}} t_3 \xrightarrow{R^*} \cdots \xrightarrow{R^*} t_{n-2} \xrightarrow{\mathsf{RW}} t_{n-1} \xrightarrow{R^*} t_n = t'$$

where $R = \mathsf{WR} \cup \mathsf{SO} \cup \mathsf{WW}$. We replace some edges from the cycle.

- First, let consider transactions t_i such that $t_i \xrightarrow{\mathsf{RW}} t_{i+1}$. This means $t_i \in \mathsf{rs}(\mathcal{K}(k,x))$ and $t_{i+1} = \mathsf{w}(\mathcal{K}(k,y))$ for some key k and two indexes x, y such that $x < y$. There are two possible cases depending on if t_i wrote the key k.
  - if t_i also wrote any key k', by Lemma 10, it also wrote the key k and we can replace the edge with a WW edge, that is $t_i \xrightarrow{\mathsf{WW}} t_{i+1}$.
  - if t_i did not wrote any key, we leave the edge the same as before.

  After the first step, any RW edge in the cycle must start from a read only transaction.

$$t_i \xrightarrow{\mathsf{RW}} t_{i+1} \Rightarrow \forall k, i.\ t \neq \mathsf{w}(\mathcal{K}(k,i)) \tag{7.12}$$

- Second, let now consider transactions t_i such that $\cdots \xrightarrow{\mathsf{RW}} t_i \xrightarrow{R^*} t_{i+1} \xrightarrow{\mathsf{RW}} \cdots$. Transaction t_i at least wrote a key but t_{i+1} is a read-only transaction, thus $t_i \neq t_{i+1}$. This means $\cdots \xrightarrow{\mathsf{RW}} t_i \xrightarrow{R^+} t_{i+1} \xrightarrow{\mathsf{RW}} \cdots$.
- Last, by Lemma 11 we replace all the WW with WR^*.

Let $R' = \mathsf{WR} \cup \mathsf{SO}$. Now we have cycle in the following form:

$$t = t'_1 \xrightarrow{R'^*} t'_2 \xrightarrow{\mathsf{RW}} t'_3 \xrightarrow{R'^+} \cdot \xrightarrow{R'^+} t'_{m-2} \xrightarrow{\mathsf{RW}} t_{m-1} \xrightarrow{R'^*} t'_m = t'$$

for some transactions t'_1 to t'_m and $m \leq n$. This means $t \xrightarrow{((\mathsf{WR} \cup \mathsf{SO});\mathsf{RW}^?)^*} t'$. Because $\mathcal{K}$ is reachable under $\mathtt{WSI}$ and so $\mathtt{CP}$, it must the case that $t \neq t'$, which contradicts with the assumption. Therefore, the relation $(\mathsf{WW} \cup \mathsf{WR} \cup \mathsf{RW} \cup \mathsf{SO})^+$ is irreflexive.

Lemma 10. *If a key-value store $\mathcal{K}$ is $\mathtt{WSI}$ safe, then for any transactions t, t'*

$$t \xrightarrow{\mathsf{RW}} t' \wedge \exists k, i.\ t = \mathsf{w}(\mathcal{K}(k, i)) \Rightarrow t \xrightarrow{\mathsf{WW}} t'$$

Proof. Assume $t \xrightarrow{\mathsf{RW}} t'$, which means $t \in \mathsf{rs}(\mathcal{K}(k, i))$ and $t' = \mathsf{w}(\mathcal{K}(k, j))$ for a key k and two indexes i, j such that $i < j$. Assume the transaction t also wrote some key k'. Since that $\mathcal{K}$ is $\mathtt{WSI}$ safe, t must write key k too, i.e. $t = \mathsf{w}(\mathcal{K}(k, z))$ for some index z. Because the $\mathcal{K}$ is reachable under $\mathtt{WSI}$ and therefore $\mathtt{UA}$, this means $z = i + 1$. Since that each version can only have one writer, we have $i < z = i + 1 < j$, therefore $t \xrightarrow{\mathsf{WW}} t'$.

Lemma 11. *If a key-value store $\mathcal{K}$ is $\mathtt{WSI}$ safe, then for any transactions t, t'*

$$t \xrightarrow{\mathsf{WW}} t' \Rightarrow t \xrightarrow{(\mathsf{WR})^*} t'$$

Proof. Assume a kv-store $\mathcal{K}$. Assume a key k and two versions of it, i and j respectively with $i < j$. Assume $t = \mathsf{w}(\mathcal{K}(k, i))$ and $t' = \mathsf{w}(\mathcal{K}(k, j))$. We prove $t \xrightarrow{(\mathsf{WR})^*} t'$ by induction on the distance of the two versions.

- Base case: $j - i = 1$. By $\mathtt{WSI}$ safe (Def. 33), t' must also read the key k, that is, $t' \in \mathsf{rs}(\mathcal{K}(k, z))$ for some z. Because the $\mathcal{K}$ is reachable under $\mathtt{WSI}$ and therefore $\mathtt{UA}$, this means that if t' read and writes key k, it must read the immediate predecessor. This means $z = i$ and then $t \xrightarrow{\mathsf{WR}} t'$.
- Inductive case: $j - i > 1$. By the Def. 33, t' must also read the key k, that is, $t' \in \mathsf{rs}(\mathcal{K}(k, z))$ for some z. Because the $\mathcal{K}$ is reachable under $\mathtt{WSI}$ and therefore $\mathtt{UA}$, this means if t' read and writes key k, it must read the immediate previous version with respect to the version it wrote. This means $z = j - 1$. Assume the writer of the z-th version is $t'' = \mathsf{w}(\mathcal{K}(k, j - 1))$. We have $t'' \xrightarrow{\mathsf{WR}} t'$. Applying I.H., we get $t \xrightarrow{(\mathsf{WR})^*} t''$. Thus we have $t \xrightarrow{(\mathsf{WR})^*} t'$.

G.3 Multiple counters

We define a multi-counter library on a set of keys $\mathtt{K}$ as the following:

$$\mathsf{Counters}(\mathsf{K}) \triangleq \bigcup_{\mathtt{k} \in \mathtt{K}} \mathsf{Counter}(k)$$

Anomaly of multiple counters under Parallel Snapshot Isolation Suppose that the kv-store contains only two keys k, k', each of which can be accessed and modified by clients using the code of transactional libraries $\mathsf{Counter}(k), \mathsf{Counter}(k'), \cdots$. We show that in this case it is possible to have the interactions of two client with the kv-store result in a non-serialisable final configuration.

More formally, suppose that $\textsc{Keys} = \{k_1, k_2\}$, and let $\mathsf{Counter} = \bigcup_{k \in \textsc{Keys}} \mathsf{Counter}(k)$. Let also

$$\begin{aligned}
\mathcal{K}_0 &= [k_1 \mapsto (0, t_0, \emptyset), k_2 \mapsto (0, t_0, \emptyset)] \\
\mathcal{K}_1 &= [k_1 \mapsto (0, t_0, \{t^1_{cl_1}\})::(1, t^1_{cl_1}, \emptyset), k_2 \mapsto (0, t_0, \emptyset)] \\
\mathcal{K}_2 &= [k_1 \mapsto (0, t_0, \{t^1_{cl_1}\})::(1, t^1_{cl_1}, \emptyset), k_2 \mapsto (0, t_0, \{t_{cl_2}\})::(1, t^1_{cl_2}, \emptyset) \\
\mathcal{K}_3 &= [k_1 \mapsto (0, t_0, \{t^1_{cl_1}\})::(1, t^1_{cl_1}, \emptyset), k_2 \mapsto (0, t_0, \{t_{cl_2}\})::(1, t^1_{cl_2}, \{t^2_{cl_1}\}) \\
\mathcal{K}_4 &= [k_1 \mapsto (0, t_0, \{t^1_{cl_1}\})::(1, t^1_{cl_1}, \{t^2_{cl_2}\}), k_2 \mapsto (0, t_0, \{t_{cl_2}\})::(1, t^1_{cl_2}, \{t^2_{cl_1}\})
\end{aligned}$$

$$\begin{aligned}
\mathcal{U}_0 &= [cl_1 \mapsto [k_1 \mapsto \{0\}, k_2 \mapsto \{0\}], cl_2 \mapsto [k_1 \mapsto \{0\}, k_2 \mapsto \{0\}]] \\
\mathcal{U}_1 &= [cl_1 \mapsto [k_1 \mapsto \{0, 1\}, k_2 \mapsto \{0\}], cl_2 \mapsto [k_1 \mapsto \{0\}, k_2 \mapsto \{0\}]] \\
\mathcal{U}_2 &= [cl_1 \mapsto [k_1 \mapsto \{0, 1\}, k_2 \mapsto \{0\}], cl_2 \mapsto [k_1 \mapsto \{0\}, k_2 \mapsto \{0, 1\}]] \\
\mathcal{U}_3 &= \mathcal{U}_2 \\
\mathcal{U}_4 &= \mathcal{U}_2
\end{aligned}$$

Observe that we have the sequence of $\mathtt{ET}_{\mathtt{PSI}}$-reductions

$$\begin{aligned}
(\mathcal{K}_0, \mathcal{U}_0) &\xrightarrow{cl_1, \mathsf{inc}(k_1)}_{\mathtt{ET}_{\mathtt{PSI}}} (\mathcal{K}_1, \mathcal{U}_1) \xrightarrow{(cl_2, \mathsf{inc}(k_2)}_{\mathtt{ET}_{\mathtt{PSI}}} \\
(\mathcal{K}_2, \mathcal{U}_2) &\xrightarrow{(cl_1, \mathsf{read}(k_2)}_{\mathtt{ET}_{\mathtt{PSI}}} (\mathcal{K}_3, \mathcal{U}_3) \xrightarrow{(cl_2, \mathsf{read}(k_1)}_{\mathtt{ET}_{\mathtt{PSI}}} (\mathcal{K}_4, \mathcal{U}_4)
\end{aligned}$$

and therefore $\mathcal{K}_4 \in \mathsf{KVStores}(\mathtt{ET}_{\mathtt{PSI}}, \mathsf{Counter})$. On the other hand, for $\mathsf{graphOf}(\mathcal{K}_4)$ we have the following cycle, which proves that $\mathcal{K}_4 \notin \mathsf{KVStrores}(\mathtt{ET}_{\mathtt{SER}}, \mathsf{Counter})$:

$$t^1_{cl_1} \xrightarrow{\mathsf{SO}_{\mathcal{K}_4}} t^2_{cl_1} \xrightarrow{\mathsf{RW}_{\mathcal{K}_4}} t^1_{cl_2} \xrightarrow{\mathsf{SO}_{\mathcal{K}_4}} t^2_{cl_2} \xrightarrow{\mathsf{RW}_{\mathcal{K}_4}} t^1_{cl_1}.$$

Robustness under Weak Snapshot Isolation It is easy to see a multi-counter libraries is $\mathtt{WSI}$ safe, therefore robust under $\mathtt{WSI}$.

Theorem 10. *Mulit-counter libraries* $\mathsf{Counters}(\mathsf{K})$ *are* $\mathtt{WSI}$ *safe.*

Proof. Assume an initial configuration $\Gamma_0 = (\mathcal{K}_0, \mathcal{U}_0$ and some $\mathtt{P}_0$ where $\mathrm{dom}[prog] \subseteq \mathrm{dom}[\mathcal{U}_0]$. Under $\mathtt{WSI}$, we prove any reachable kv-store $\mathcal{K}_i$ satisfies Eqs. (7.7) to (7.9) by induction on the length of trace.

- Base case: $i = 0$. The formulae Eqs. (7.7) to (7.9) trivially hold given $\mathcal{K}_0$ contains only the initial transaction t_0.

– Inductive case: $i > 0$. Let $\Gamma_i = (\mathcal{K}_i, \mathcal{U}_i)$ be the result of running $\mathtt{P}_0$ for i steps. We perform case analysis for the next transaction t_{i+1}.
  - If t_{i+1} reads a key $\mathtt{k}$, i.e. $\mathsf{read}(\mathtt{k})$, it must start from a view that is closed to the relation $((\mathsf{WW} \cup \mathsf{WR} \cup \mathsf{SO}) \cup \mathsf{WR};\mathsf{RW} \cup \mathsf{SO};\mathsf{RW})^*$. Let $\mathcal{K}_i(\mathtt{k}, j) = (v, t', T')$ be the latest version included in the view. Thus the new kv-store $\mathcal{K}_{i+1} = \mathcal{K}_i[\mathtt{k} \mapsto \mathcal{K}_i(\mathtt{k})[j \mapsto (v, t', T' \uplus \{t_{i+1}\})]]$. Given t_{i+1} only read the key $\mathtt{k}$ without writing, Eqs. (7.7) to (7.9) trivially holds. For other transactions t that are different from t_{i+1}, they must exist in $\mathcal{K}_i$. By I.H., then we prove that $mkvs_{i+1}$ is $\mathtt{WSI}$ safe.
  - If t_{i+1} increments a key $\mathtt{k}$, i.e. $\mathsf{inc}(\mathtt{k})$, it means that all versions of k must be included in the view. Let $\mathcal{K}_i(\mathtt{k}, j) = (v, t', T')$ be the latest version of key $\mathtt{k}$. Thus the new kv-store $\mathcal{K}_{i+1} = \mathcal{K}_i[\mathtt{k} \mapsto (\mathcal{K}_i(\mathtt{k})[j \mapsto (v, t', T' \uplus \{t_{i+1}\})])::(v+1, t_{i+1}, \emptyset)]$. Given t_{i+1} only read and then rewrites the key $\mathtt{k}$, Eqs. (7.7) to (7.9) trivially holds. For other transactions t that are different from t_{i+1}, they must exist in $\mathcal{K}_i$. By I.H., then we prove that $mkvs_{i+1}$ is $\mathtt{WSI}$ safe.

G.4 Bank Example

Alomari et al. [3] presented a bank example and claimed that this example is robust against $\mathtt{SI}$. We find out that the bank example is also robust against $\mathtt{WSI}$. The example bases on relational database with three tables, account, saving and checking. The account table maps customer names to customer IDs ($\mathtt{Account}(\underline{\mathtt{Name}}, \mathtt{CustomerID})$) and saving and checking map customer IDs to saving balances ($\mathtt{Saving}(\underline{\mathtt{CustomerID}}, \mathtt{Balance})$) and checking balances ($\mathtt{Checking}(\underline{\mathtt{CustomerID}}, \mathtt{Balance})$) respectively. We ignore the account table since it is an immutable lookup table. We encode the saving and checking tables together as a kv-store. Each customer is represent as an integer n, that is, $(\_, n) \in \mathtt{Account}(\underline{\mathtt{Name}}, \mathtt{CustomerID})$, its checking balance is associated with key $n_s = 2 \times n$ and saving with $n_c = 2 \times n + 1$.

$$n_c \triangleq 2 \times n \qquad n_s \triangleq 2 \times n + 1 \qquad \textsc{Keys} \triangleq \bigcup_{n \in \mathbb{N}} \{n_c, n_s\}$$

If n is a customer, then

$$(n, \mathsf{val}(\mathcal{K}(n_s, |\mathcal{K}(n_s)|))) \in \mathtt{Saving}(\underline{\mathtt{CustomerID}}, \mathtt{Balance})$$

and

$$(n, \mathsf{val}(\mathcal{K}(n_s, |\mathcal{K}(n_c)|))) \in \mathtt{Checking}(\underline{\mathtt{CustomerID}}, \mathtt{Balance})$$

To interact with tables, there are five types of transactions. For brevity we assume balances are integers.

$$\begin{aligned}
\mathtt{balance}(\mathtt{n}) &\triangleq [\mathtt{x} := [\mathtt{n}_c]\,;\ \mathtt{y} := [\mathtt{n}_s]\,;\ \mathtt{ret} := \mathtt{x} + \mathtt{y}] \\
\mathtt{depositChecking}(\mathtt{n}, \mathtt{v}) &\triangleq [\mathtt{if}\ (\mathtt{v} \geq 0)\{\ \mathtt{x} := [\mathtt{n}_c]\,;\ [\mathtt{n}_c] := \mathtt{x} + \mathtt{v};\ \}] \\
\mathtt{transactSaving}(\mathtt{n}, \mathtt{v}) &\triangleq [\mathtt{x} := [\mathtt{n}_s]\,;\ \mathtt{if}\ (\mathtt{v} + \mathtt{x} \geq 0)\{\ [\mathtt{n}_s] := \mathtt{x} + \mathtt{v};\ \}]
\end{aligned}$$

$\mathtt{balance}(\mathtt{n})$ returns customer n total balance. $\mathtt{depositChecking}(\mathtt{n}, \mathtt{v})$ deposits v to the checking account of customer n, if v is non-negative, otherwise the

transaction does nothing. While $\texttt{transactSaving}(\texttt{n}, \texttt{v})$ allows a consumer n to deposit or withdraw money from the saving account as long as the saving account afterwards is non-negative.

$$\texttt{amalgamate}(\texttt{n}, \texttt{n}') \triangleq \begin{bmatrix} \texttt{x} := [\texttt{n}_s]\,;\ \texttt{y} := [\texttt{n}_c]\,;\ \texttt{z} := [\texttt{n}'_c]\,; \\ [\texttt{n}_s] := 0;\ [\texttt{n}_c] := 0;\ [\texttt{n}'_c] := \texttt{x} + \texttt{y} + \texttt{z}; \end{bmatrix}$$

$$\texttt{writeCheck}(\texttt{n}, \texttt{v}) \triangleq \begin{bmatrix} \texttt{x} := [\texttt{n}_s]\,;\ \texttt{y} := [\texttt{n}_c]\,; \\ \texttt{if}\ (\texttt{x} + \texttt{y} < \texttt{v})\{\ [\texttt{n}_c] := \texttt{y} - \texttt{v} - 1;\ \}\texttt{else}\{\ [\texttt{n}_c] := \texttt{y} - \texttt{v};\ \} \\ [\texttt{n}_s] := \texttt{x}; \end{bmatrix}$$

$\texttt{amalgamate}(\texttt{n}, \texttt{n}')$ represents moving all funds from consumer n to the checking account of customer n'. Last, $\texttt{writeCheck}(\texttt{n}, \texttt{v})$ updates the checking account of n. If funds, both saving and checking, from n is greater than the v, the transaction deduct v from the checking account of n. If funds are not enough, the transaction further deducts one pounds as penalty. Alomari et al. [3] argued that, to make this example robust against `SI`, $\texttt{writeCheck}(\texttt{n}, \texttt{v})$ must be strengthened by writing back the balance to the saving account (the last line, $[\texttt{n}_s] := \texttt{x}$), even thought the saving balance is unchanged. The bank `bank` libraries are defined by

$$\texttt{Bank} \triangleq \left\{ \begin{array}{r} \texttt{balance}(\texttt{n}), \texttt{depositChecking}(\texttt{n}, \texttt{v}), \texttt{amalgamate}(\texttt{n}, \texttt{n}'), \\ \texttt{writeCheck}(\texttt{n}, \texttt{v}), \texttt{writeCheck}(\texttt{n}, \texttt{v}) \end{array} \;\middle|\; n, n' \in \mathbb{N} \wedge v \in \mathbb{Z} \right\}$$

Theorem 11. *The bank libraries* `Bank` *are* `WSI` *safe.*

Proof. Assume an initial configuration $\Gamma_0 = (\mathcal{K}_0, \mathcal{U}_0)$ and some $\texttt{P}_0$ where $\mathrm{dom}(\texttt{P}) \subseteq \mathrm{dom}(\mathcal{U}_0)$. Under `WSI`, we prove any reachable kv-store $\mathcal{K}_i$ satisfies Eqs. (7.7) to (7.9) by induction on the length of trace.

- Base case: $i = 0$. The formulae Eqs. (7.7) to (7.9) trivially hold given $\mathcal{K}_0$ contains only the initial transaction t_0.
- Inductive case: $i > 0$. Let $\Gamma_i = (\mathcal{K}_i, \mathcal{U}_i)$ be the result of running $\texttt{P}_0$ for i steps. We perform case analysis for the next transaction t_{i+1}.
  - If t_{i+1} is $\texttt{balance}(\texttt{n})$, the only possible fingerprint is $\{(\texttt{r}, n_c, v_c), (\texttt{r}, n_s, v_s)\}$ for some values v_c and v_s. Since it is a read-only transaction, Eqs. (7.7) to (7.9) trivially hold.
  - If t_{i+1} is $\texttt{depositChecking}(\texttt{n}, \texttt{v})$, in the cases of $v < 0$, the fingerprint is empty and Eqs. (7.7) to (7.9) trivially hold. However, in the case of $v \geq 0$, the fingerprint is $\{(\texttt{r}, n_c, v_c), (\texttt{w}, n_c, v_c + v)\}$. Because it read and wrote only one key, n_c, Eqs. (7.7) to (7.9) hold.
  - If t_{i+1} is $\texttt{transactSaving}(\texttt{n}, \texttt{v})$, there are two cases: either a read-only fingerprint $\{(\texttt{r}, n_s, v_s)\}$ when saving account has insufficient funds, or a read and write on key n_s, that is $\{(\texttt{r}, n_s, v_s), (\texttt{w}, n_s, v_s + v)\}$. For both cases it is easy to see Eqs. (7.7) to (7.9) hold.

- If t_{i+1} is $\texttt{amalgamate}(\texttt{n}, \texttt{n}')$, the fingerprint is

  $$\{(\texttt{r}, n_s, v_s), (\texttt{w}, n_s, 0), (\texttt{r}, n_c, v_c), (\texttt{w}, n_c, 0), (\texttt{r}, n'_c, v'_c), (\texttt{w}, n'_c, v'_c + v_s + v_c)\}$$

  Because the transaction always read and then wrote keys it touched, namely n_s, n_c and n'_c, Eqs. (7.7) to (7.9) hold.
- Last, if t_{i+1} is $\texttt{writeCheck}(\texttt{n}, \texttt{v})$, the fingerprint is

  $$\{(\texttt{r}, n_s, v_s), (\texttt{w}, n_s, v_s), (\texttt{r}, n_c, v_c), (\texttt{w}, n_c, v'_c)\}$$

  where v'_c can be either $v_c - v$ or $v_c - v - 1$. Similar to $\texttt{amalgamate}(\texttt{n}, \texttt{n}')$, the transaction always read and then wrote keys it touched, so Eqs. (7.7) to (7.9) hold.

H Verification of implementations

We verify two protocols, COPS and Clock-SI, that the former is a full replicated implementation for causal consistency and the latter is a shard implementation for snapshot isolation.

H.1 COPS

In § 5, we have explained the COPS protocol that retracts transactions to be single-write and multiple-read[4] and implements a replicated database that satisfies causal consistency. In this section, we give a formal model and semantics for COPS protocol and verify the COPS protocol via trace refinement with respect to our operational semantics.

Notation. In this section, all the variables related to COPS protocol are annotated with *hat*, for example $\hat{t}$. We pick the same letters for the concepts that have similar meaning to our operational model.

Reference Machine States In COPS protocol, there is no explicit transactional identifier, but the version identifier, which consist of the replica identifier who initially accepted the version and the local time when the version was accepted, can be seen as the transactional identifier who wrote the version.

Definition 34 (COPS Replica and Transaction Identifiers). *The set of* COPS replicas, $\text{COPSReps} \ni r$, *is a countably infinite set that is totally ordered on relation* $\leq$. *The set of* COPS transaction identifiers, $\text{COPSTxIDs} \ni \hat{t}$, *is defined by*

$$\text{COPSTxIDs} \stackrel{def}{=} \left\{ \hat{t}_{cl}^{(n,r,m)} \;\middle|\; cl \in \text{ClientID} \wedge r \in \text{COPSReps} \wedge n, m \in \mathbb{N} \right\} \cup \{t_0\}.$$

The order over transactions is defined by

$$\hat{t}_{cl}^{(n,r,m)} \sqsubseteq \hat{t}_{cl'}^{(n',r',m')} \stackrel{def}{\Leftrightarrow} n < n' \vee (n = n' \wedge (r < r' \vee (r = r' \wedge m < m')).$$

We annotate the COPS transaction identifier $\hat{t}_{cl}^{(n,r,m)}$ with the client identifier cl, the local time n, the replica identifier r and the extra counter m for read-only transactions. In the model and semantics of COPS protocol (§H.1), the extra counter m is always ZERO. Later in the verification of COPS protocol (§H.1), we use the extra counter m to identify read-only transactions. Note that, all transactions, including read-only transactions, are totally order by (n, r, m) lexicographically.

As explained before, a COPS version $\hat{\nu}$ consists of the value, the identifier and the dependency set that is other versions $\hat{\nu}$ depends on.

Definition 35 (COPS versions and dependency sets). *A* dependency set $\hat{d}$ *is defined by* $\hat{d} \subseteq \text{Keys} \times \text{COPSTxIDs}$. *Let* $\text{COPSDeps} \ni \hat{d}$ *denote the set of all dependency sets. The set of* COPS versions $\text{COPSVers} \ni \hat{\nu}$, *is defined by* $\text{COPSVers} \stackrel{def}{=} \text{Value} \times \text{COPSTxIDs} \times \text{COPSDeps}$. *Let* $\mathsf{ValueOf}(\hat{\nu})$, $\mathsf{WriterOf}(\hat{\nu})$ *and* $\mathsf{DepSetOf}(\hat{\nu})$ *return the first to third projections of* $\hat{\nu}$ *respectively.*

[4] A simplified COPS protocol only allows single-write and single-read transactions

Each replica r consists of a local multi-versioning key-value store and a local time. A COPS database contains finite number of replicas that are opaque to the clients.

Definition 36 (COPS Key-value Stores and Databases). *the set of* COPS local key-value stores *or* COPS local stores *is defined by*

$$\text{COPSKVSs} \stackrel{def}{=} \left\{ \hat{\mathcal{K}} \in \textsc{Keys} \to [\text{COPSVers}] \;\middle|\; \texttt{WfCOPSKvs}\left(\hat{\mathcal{K}}\right) \right\}.$$

where `WfCOPSKvs` *is defined by: for any keys* $k, k' \in \textsc{Keys}$*, indexes* $i, i' \in \mathbb{N}$ *and the initial value* v_0*,*

$$\hat{\mathcal{K}}(k, 0) = (v_0, t_0, \_), \tag{8.1}$$

$$\mathsf{WriterOf}\left(\hat{\mathcal{K}}(k, i)\right) = \mathsf{WriterOf}\left(\hat{\mathcal{K}}(k', i')\right) \Rightarrow k = k' \wedge i = i', \tag{8.2}$$

$$\mathsf{WriterOf}\left(\hat{\mathcal{K}}(k, i)\right) \sqsubseteq \mathsf{WriterOf}\left(\hat{\mathcal{K}}(k, i')\right) \Leftrightarrow i < i'. \tag{8.3}$$

Given two COPS local stores $\hat{\mathcal{K}}, \hat{\mathcal{K}}'$*, the order* $\hat{\mathcal{K}} \sqsubseteq \hat{\mathcal{K}}'$ *is defined by*

$$\hat{\mathcal{K}} \sqsubseteq \hat{\mathcal{K}}' \stackrel{def}{\Leftrightarrow} \forall k \in \textsc{Keys}.\, \forall i \in \mathbb{N}.\, \hat{\nu} = \hat{\mathcal{K}}(k, i) \Rightarrow \exists i \in \mathbb{N}.\, \hat{\nu} = \hat{\mathcal{K}}'(k, i').$$

The set of COPS databases *is defined by*

$$\text{COPSs} \stackrel{def}{=} \left\{ \begin{array}{l|l} \mathcal{R} \in \text{COPSReps} \stackrel{fin}{\rightharpoonup} & \forall r, r' \in \mathsf{Dom}(\mathcal{R}).\, \forall k, k' \in \textsc{Keys}.\, \forall i, i' \in \mathbb{N}. \\ \quad (\text{COPSKVSs} \times \mathbb{N}) & \mathsf{WriterOf}(\mathcal{R}(r)(k, i)) = \mathsf{WriterOf}(\mathcal{R}(r')(k', i')) \Leftrightarrow \\ & \qquad k = k' \wedge \mathcal{R}(r)(k, i) = \mathcal{R}(r')(k', i') \end{array} \right\}.$$

A COPS replica tracks all the history versions that are uniquely identified and ordered by their writers. Note that, without losing generality, versions for a key are organised into a list; the relative position of versions does not matter since all versions are ordered by their writers. Since versions will be eventually delivered to all sites, this means that versions with the same writer from different replicas must be the same version, which is captured by the well-formed condition `WfCOPSKvs`.

Each COPS client maintain a local context to trace versions that the client read or wrote before. The client contexts is used to build the correct dependency sets for versions. Each client always interacts with a assigned replica (identifier).

Definition 37 (COPS Client Contexts and Environments). *A* COPS context $\hat{u}$ *is defined by* $\hat{u} \subseteq \textsc{Keys} \times \text{COPSTxIDs}$*. Let* $\text{COPSCtxs} \in \hat{u}$ *denotes the set of* COPS contexts*. The set of* COPS client contexts $\text{COPSCtxEnvs} \ni \mathcal{C}$ *is defined by* $\text{COPSCtxEnvs} \stackrel{def}{=} \textsc{ClientID} \stackrel{fin}{\rightharpoonup} (\text{COPSCtxs} \times \text{COPSReps})$.

When a client commits a new version, the client context becomes the dependency set of the new version. Thus both dependency sets of versions and client contexts are modelled as sets of tuples with keys and identifiers.

Last, a COPS configuration consists of a COPS database $\mathcal{R}$ and a COPS client environment $\hat{\mathcal{C}}$. Any client from the environment $\hat{\mathcal{C}}$ must be served by a replica that exists in $\mathcal{R}$.

Definition 38 (COPS Configurations). *A COPS configuration* $\Theta \in \text{COPSConfs}$ *comprises a COPS database* $\mathcal{R}$ *and a client context* $\mathcal{C}$*, that is,*

$$\text{COPSConfs} \stackrel{def}{=} \left\{ \begin{array}{l|l} (\mathcal{R}, \mathcal{C}) \in & \forall cl \in \mathsf{Dom}\left(\mathcal{C}\right). \forall r \in \text{COPSReps}. \\ \quad \text{COPSs} \times \text{COPSCtxEnvs} & \quad \mathcal{C}\left(cl\right) = (\_, r) \Rightarrow r \in \mathsf{Dom}\left(\mathcal{R}\right) \end{array} \right\}.$$

The set of initial COPS configurations, $\text{COPSConfs}_0 \ni \hat{\Gamma}_0$*, is defined by*

$$\text{COPSConfs}_0 \stackrel{def}{=} \left\{ \begin{array}{l|l} (\mathcal{R}, \mathcal{C}) \in & \forall r \in \text{COPSReps}. \forall cl \in \text{ClientID}. \\ \quad \text{COPSConfs} & \quad \mathcal{R}\left(r\right) = \lambda k \in \text{Keys}. \left([(v_0, t_0, \emptyset)], 0\right) \wedge \mathcal{C}\left(cl\right) = (\emptyset, \_) \end{array} \right\}.$$

Reference Operational Semantics A COPS client interacts with the replica via calling `put` and `read`: (1) function $\texttt{put}\left(k, v\right)$ commits a new value v for key k, and the replica will accept this version immediately and then broadcast to other replicas via synchronised messages; (2) function $\texttt{read}\left(K\right)$ requests a causally dependent snapshot for the key set K, and for better performance, the replica will prepare this snapshot in a fine-grained manner, reading keys one by one. The API is modelled in Def. 39. To models the fine-grained read in the replica side, we introduce *runtime commands* COPSRuntimeCmds to track intermediate states of `read`.

Definition 39 (COPS Commands and Programs). *A* COPS command $\hat{\texttt{C}}$ *is defined by the following grammar: for any key* $k \in \text{Keys}$*, key set* $K \subseteq \text{Keys}$ *and* $v \in \text{Value}$*,*

$$\hat{\texttt{C}} ::= \texttt{put}\left(k, v\right) | \texttt{read}\left(K\right) | \hat{\texttt{C}}\,;\hat{\texttt{C}}.$$

Let $\text{COPSCmds} \ni \hat{\texttt{C}}$ *denote the set of* COPS commands*. A* COPS runtime command $\hat{\texttt{R}}$ *is defined by the following grammar: for any* $\hat{\mathcal{V}}, \hat{\mathcal{V}}' \in [\text{COPSVers}]$ *and* $\hat{D} \in [\text{COPSDeps}]$*,*

$$\hat{\texttt{R}} ::= \texttt{put}\left(k, v\right) | \texttt{read}\left(K\right) | \texttt{read}\left(K\right) : \hat{\mathcal{V}} | \texttt{read}\left(K\right) : (\hat{\mathcal{V}}, \hat{D}) | \texttt{read}\left(K\right) : (\hat{\mathcal{V}}, \hat{D}, \hat{\mathcal{V}}') | \hat{\texttt{R}};\hat{\texttt{R}}.$$

Let $\text{COPSRuntimeCmds} \ni \hat{\texttt{R}}$ *denote the set of* COPS runtime commands*. The sets of* COPS programs, $\text{COPSProgs} \ni \hat{\texttt{P}}$*, and* COPS runtime programs, $\text{COPSRuntimeProgs} \ni \hat{\texttt{I}}$*, are defined by* $\text{COPSProgs} \stackrel{def}{=} \text{ClientID} \xrightarrow{fin} \text{COPSCmds}$ *and* $\text{COPSRuntimeProgs} \stackrel{def}{=} \text{ClientID} \xrightarrow{fin} \text{COPSRuntimeCmds}$ *respectively.*

A replica prepares the causally dependent snapshot for a key set K in three phases, $\texttt{read}\left(K\right) : \hat{\mathcal{V}}$, $\texttt{read}\left(K\right) : (\hat{\mathcal{V}}, \hat{D})$ and $\texttt{read}\left(K\right) : (\hat{\mathcal{V}}, \hat{D}, \hat{\mathcal{V}}')$. We will give the semantics later.

Proposition 20 (COPS Runtime Command Inclusion). $\text{COPSCmds} \subseteq \text{COPSRuntimeCmds}$.

We verify COPS via trace refinement. To help the proofs, we label each step in the COPS trace: each label corresponds to a rule in the semantics (Figs. 11a, 12a and 13a). Any local computation is labelled with $(cl, r, \bullet)$. The reduction step for a single-write transaction is labelled with $\left(cl, r, (\mathtt{w}, k, v), \hat{t}, \hat{d}\right)$. As explained

before, multiple-read requests are processed in two phase internally in the replica. The labels $\left(cl, r, (\mathtt{r}, k, v), \hat{t}, \hat{d}, \dagger\right), (cl, r, \dagger), \left(cl, r, (\mathtt{r}, k, v), \hat{t}, \hat{d}, \ddagger\right), (cl, r, \hat{u}, \ddagger)$ means optimistic read of k, phase change, re-fetch of k and return the set of versions $\hat{u}$ respectively. Last, $(r, \hat{t})$ labels the step where replica r receives new version $\hat{\nu}$.

The reference semantics for COPS commands is defined in Figs. 11a and 12a. Rules are labelled with information that are usefully for proving properties.

Definition 40 (COPS Labels). *The set of* COPS labels $\textsc{Labels} \ni \iota$ *is defined by*

$$\textsc{Labels} \supseteq \bigcup_{\substack{cl \in \textsc{ClientID}, r \in \textsc{COPSReps} \\ k \in \textsc{Keys}, v \in \textsc{Value} \\ \hat{t} \in \textsc{COPSTxIDs}, \hat{d} \in \textsc{COPSDeps}}} \left\{ \begin{array}{c} (cl, r, \bullet), \left(cl, r, (\mathtt{w}, k, v), \hat{t}, \hat{d}\right), \\ \left(cl, r, (\mathtt{r}, k, v), \hat{t}, \hat{d}, \dagger\right), (cl, r, \dagger), \\ \left(cl, r, (\mathtt{r}, k, v), \hat{t}, \hat{d}, \ddagger\right), (cl, r, \hat{u}, \ddagger), (r, \hat{t}) \end{array} \right\}.$$

Rule for `put`. When a client calls `put` function (Fig. 11b), it sends a new value `v` for key `k` with the context $\hat{u}$ to a replica r. Once the replica receives the update request, it creates a new version for `k` with the new value `v`, allocates this version with a new version identifier consisting of the next local time-stamp and the replica identifier and assigns the client context `ctx` as the dependency set (line 7 in Fig. 11b). This new version will be added into the context in line 8 in Fig. 11b. Now the replica is ready to send back the acknowledgement to the client. Last, the new version is broadcasted to all other replicas.

$$\textsc{COPSWrite}\ \frac{\hat{t} = \hat{t}_{cl}^{(n+1,r,0)} \qquad \hat{\nu} = (v, \hat{t}, \hat{u}) \qquad \hat{\mathcal{K}}' = \mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k, \hat{\nu}\right) \qquad \hat{u}' = \hat{u} \uplus \left\{(k, \hat{t})\right\}}{\left(\hat{\mathcal{K}}, \hat{u}, n\right), \mathtt{put}\,(k, v) \xrightarrow{\left(cl, r, (\mathtt{w}, k, v), \hat{t}, \hat{u}\right)} \left(\hat{\mathcal{K}}', \hat{u}', n+1\right), \mathtt{skip}}$$

(a) The reference semantics for `put`

```
1  // mixing the client API and system API
2  put(repl,k,v,ctx) {
3      deps = ctx; // Dependency set as previous reads and
           writes
4      // increase local time and appending local kv with a new
           version.
5      ltime = inc(repl.local_time);
6      txid = (ltime ++ repl.id);
7      list_isnert(repl.kv[k],(v, txid, deps));
8      ctx += (k, txid); // the new id put into client context
9      asyn_brordcase(k, v, txid, deps); // broad case
10 }
```

(b) The reference implementation for `put`

Fig. 11: COPS API: `put`

In COPSWrite rule in Fig. 11a, the second and third premises model lines 5 and 7 in Fig. 11b respectively. Function $\mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k, \hat{\nu}\right)$ inserts the version in a position that preserves the order.

Definition 41 (List Insertion). *Given a COPS store* $\hat{\mathcal{K}} \in \text{COPSKVSs}$ *and a version* $\hat{\nu} \in \text{COPSVers}$ *for a key* $k \in \text{Keys}$. *Function* $\mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k, \hat{\nu}\right)$ *is defined by*

$$\begin{aligned}\mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k, \hat{\nu}\right) \stackrel{def}{=} & \texttt{let } \hat{\mathcal{V}} = \hat{\mathcal{K}}\left(k\right) \texttt{ and } \left[\hat{\nu}_0, \cdots, \hat{\nu}_i, \hat{\nu}_{i+1}, \cdots, \hat{\nu}_n\right] = \hat{\mathcal{V}} \\ & \texttt{where } \mathsf{WriterOf}\left(\hat{\nu}_i\right) \sqsubseteq \mathsf{WriterOf}\left(\hat{\nu}\right) \sqsubseteq \mathsf{WriterOf}\left(\hat{\nu}_{i+1}\right) \\ & \texttt{in } \hat{\mathcal{K}}\left[k \mapsto \left[\hat{\nu}_0, \cdots, \hat{\nu}_i, \hat{\nu}, \hat{\nu}_{i+1}, \cdots, \hat{\nu}_n\right]\right].\end{aligned}$$

Proposition 21 (Well-defined COPSInsert). *Given a COPS store* $\hat{\mathcal{K}} \in \text{COPSKVSs}$ *and a version* $\hat{\nu} \in \text{COPSVers}$ *for a key* $k \in \text{Keys}$ *such that* $\mathsf{WriterOf}\left(\hat{\nu}\right) \notin \hat{\mathcal{K}}$, *the new store* $\mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k, \hat{\nu}\right)$ *is a well-formed COPS store.*

Rules for `read`. Fig. 12b presents the reference implementation for `read` that a client requests a snapshot for keys `ks` from a replica r. The semantics for `read` is given in Fig. 12a. As explained before, the replica constructs this snapshot in two phases.

In the first phase (line 3 in Fig. 12b), replica reads the latest version for each key (using `get_by_version`, with `LATEST` label). However, interleaving may happen between two reads. COPSOptRead rule models this phase: the latest version $\hat{\nu}_i$ for key k_i in the replica kv-store $\hat{\mathcal{K}}$ is added to the runtime version lists $\hat{\mathcal{V}}$, that is, $\hat{\mathcal{V}} :: [\hat{\nu}_i]$. The label for this rule tracks the client, the replica, the version and a † to indicate the first phase.

At the end of the first phase, replica collects all dependency sets and uses them as an lower-bound in the second phase. This lower bound contains minimum versions in term of the version identifiers that can form a causal dependent snapshot for the key set K. In COPSLowerBound rule, we use $\mathsf{VerLower}\left(K, \hat{\mathcal{V}}\right)$ function to model the computation shown in line 9.

Definition 42 (Version Lower-bound). *Given a key set* K *and list of versions for the key set* $\hat{\mathcal{V}}$ *the function* $\mathsf{VerLower}\left(K, \hat{\mathcal{V}}\right)$ *is defined by*

$$\mathsf{VerLower}\left(K, \hat{\mathcal{V}}\right) \stackrel{def}{=} \texttt{let } n = |K| - 1 \texttt{ in } \left[\mathsf{VerLowerN}\left(K_{|0}, \hat{\mathcal{V}}, 0\right), \cdots, \mathsf{VerLowerN}\left(K_{|n}, \hat{\mathcal{V}}, n\right)\right]$$

$$\mathsf{VerLowerN}\left(k, \hat{\mathcal{V}}, i\right) \stackrel{def}{=} \left(k, \mathsf{Max}_{\sqsubseteq}\left(\left\{\hat{t} \;\middle|\; \hat{t} = \mathsf{WriterOf}\left(\hat{\mathcal{V}}_{|i}\right) \vee \exists i'.\, (k, \hat{t}) \in \mathsf{DepSetOf}\left(\hat{\mathcal{V}}_{|i'}\right)\right\}\right)\right).$$

The second phase re-fetches the version $\hat{\nu}$ for each key k contained in the lower-bound, if the version that was read in the first phase is older than $\hat{\nu}$, which is captured by line 13 in Fig. 12b. We model this second phase by COPSRefetch rule: it re-fetches the version contained in the lower-bound $\hat{D}$ if the

COPSStartRead

$$\frac{}{\left(\hat{\mathcal{K}}, \hat{u}, n\right), \texttt{read}\,(K) \xrightarrow{(cl,r,\bullet)} \left(\hat{\mathcal{K}}, \hat{u}, n\right), \texttt{read}\,(K) : \emptyset}$$

COPSOptRead

$$\frac{\texttt{Unique}\,(K) \qquad K = [k_0, \cdots, k_i, \cdots, k_m] \qquad \hat{\mathcal{V}} = [\hat{\nu}_0, \cdots, \hat{\nu}_{i-1}] \qquad \hat{\mathcal{K}}\left(k, \left|\hat{\mathcal{K}}\,(k)\right| - 1\right) = (v_i, \hat{t}_{cl_i}^{(n_i,r_i,0)}, \hat{d}_i) = \hat{\nu}_i \qquad \iota = \left(cl, r, (\texttt{r}, k_i, v_i), \hat{t}_{cl_i}^{(n_i,r_i,0)}, \hat{d}_i, \dagger\right)}{\left(\hat{\mathcal{K}}, \hat{u}, n\right), \texttt{read}\,(K) : \hat{\mathcal{V}} \overset{\iota}{\rightarrow} \left(\hat{\mathcal{K}}, \hat{u}, n\right), \texttt{read}\,(K) : (\hat{\mathcal{V}} :: [\hat{\nu}_i])}$$

COPSLowerBound

$$\frac{|K| = \left|\hat{\mathcal{V}}\right| \qquad \hat{D} = \mathsf{VerLower}\left(K, \hat{\mathcal{V}}\right)}{\left(\hat{\mathcal{K}}, \hat{u}, n\right), \texttt{read}\,(K) : \hat{\mathcal{V}} \xrightarrow{(cl,r,\dagger)} \left(\hat{\mathcal{K}}, \hat{u}, n\right), \texttt{read}\,(K) : (\hat{\mathcal{V}}, \hat{D})}$$

COPSRefetch

$$\frac{\hat{\mathcal{V}}' = [\hat{\nu}_0, \cdots, \hat{\nu}_{i-1}] \qquad \hat{D}_{|i} = (k_i, \hat{t}) \qquad (v_i, \hat{t}_i, \hat{d}_i) = \begin{cases} \hat{\mathcal{K}}\,(k_i, m) & \text{if } \mathsf{WriterOf}\left(\hat{\mathcal{V}}_{|i}\right) \sqsubseteq \hat{t} \wedge \exists m.\, \hat{t} = \mathsf{WriterOf}\left(\hat{\mathcal{K}}\,(k_i, m)\right) \\ \hat{\mathcal{V}}_{|i} & \text{otherwise} \end{cases} \qquad \hat{\mathcal{V}}'' = \hat{\mathcal{V}}' :: \left[(v_i, \hat{t}_i, \hat{d}_i)\right] \qquad \iota' = \left(cl, r, (\texttt{r}, k_i, v_i), \hat{t}_i, \hat{d}_i, \ddagger\right)}{\left(\hat{\mathcal{K}}, \hat{u}, n\right), \texttt{read}\,(K) : (\hat{\mathcal{V}}, \hat{D}, \hat{\mathcal{V}}') \overset{\iota'}{\rightarrow} \left(\hat{\mathcal{K}}, \hat{u}, n\right), \texttt{read}\,(K) : (\hat{\mathcal{V}}, \hat{D}, \hat{\mathcal{V}}'')}$$

COPSFinishRead

$$\frac{|K| = \left|\hat{\mathcal{V}}'\right| \qquad \hat{u}' = \left\{(k, \hat{t}) \;\middle|\; (k, \hat{t}, \_) \in \hat{\mathcal{V}}'\right\}}{\left(\hat{\mathcal{K}}, \hat{u}, n\right), \texttt{read}\,(K) : (\hat{\mathcal{V}}, \hat{D}, \hat{\mathcal{V}}') \xrightarrow{(cl,r,\hat{u}',\ddagger)} \left(\hat{\mathcal{K}}, \hat{u} \cup \hat{u}', n\right), \texttt{skip}}$$

(a) The reference semantics for `read`

```
1  List(Val) read(repl,ks,ctx) {
2      // Only guarantee to read up-to-date value at the moment
3      for k in ks { rst[k] = get_by_version(repl,k,LATEST); }
4      // initially the list of lower bound for keys
5      for k in ks { ccv[k] = rst[k].id;  }
6      // compute the lower bound
7      for k in ks {
8          for dep in rst[k].deps
9              if ( dep.key ∈ ks ) ccv[k] = max (ccv[dep.key].id
                   , dep.id);
10     }
11     // re-fetch a new version for some key
12     for k in ks
13         if ( ccv[k] > rst[k].vers ) rst[k] = get_by_version(k
               ,ccv[k]);
14     // update the ctx
15     for (k,ver,deps) in rst { ctx.readers += (k,ver,deps); }
16     return to_vals(rst);
17 }
```

(b) The reference implementation for `read`

Fig. 12: COPS API: `read`

version read in the first phase, that is, $\hat{\mathcal{V}}_{|i}$, is older than the version contained in the lower-bound, that is, $copsdepset_{|i}$.

Last, the replica returns the list of versions for the key set K to the client and the client must include the versions in its context, which is captured by line 15 and modelled by COPSFinishRead rule.

All the rules for read and write are lift to a program level standard interleaving semantics presented in Fig. 13a: COPSClient rule models that a client takes steps in turn.

$\rightarrow$: COPSConfs × COPSRuntimeProgs × Labels × COPSConfs × COPSRuntimeProgs

COPSClient

$$\frac{\mathcal{R}(r) = (\hat{\mathcal{K}}, n) \qquad \mathcal{C}(cl) = (\hat{u}, r) \qquad \hat{\mathtt{I}}(cl) = \hat{\mathtt{R}} \qquad \left(\hat{\mathcal{K}}, \hat{u}, n\right), \hat{\mathtt{R}} \overset{\downarrow}{\rightarrow} \left(\hat{\mathcal{K}}', \hat{u}', n'\right), \hat{\mathtt{R}}' \qquad \mathcal{R}' = \mathcal{R}\left[r \mapsto (\hat{\mathcal{K}}, n')\right] \qquad \mathcal{C}' = \mathcal{C}\left[cl \mapsto (\hat{u}', r)\right] \qquad \hat{\mathtt{I}}' = \hat{\mathtt{I}}\left[cl \mapsto \hat{\mathtt{R}}\right]}{(\mathcal{R}, \mathcal{C}), \hat{\mathtt{I}} \overset{\downarrow}{\rightarrow} \left(\mathcal{R}', \mathcal{C}'\right), \hat{\mathtt{I}}'}$$

COPSSync

$$\frac{r \neq r' \qquad \mathcal{R}(r) = (\hat{\mathcal{K}}, n) \qquad \mathcal{R}\left(r'\right) = (\hat{\mathcal{K}}', n') \qquad \hat{\mathcal{K}}(k, i) = \hat{\nu} \qquad \forall \hat{t}.\, (\_, \hat{t}) \in \mathsf{DepSetOf}\,(\hat{\nu}) \Rightarrow \hat{t} \in \hat{\mathcal{K}}' \qquad \hat{\mathcal{K}}'' = \mathsf{COPSInsert}\left(\hat{\mathcal{K}}', k, \hat{\nu}\right) \qquad \hat{t}_{\_}^{(n'', \_, 0)} = \mathsf{WriterOf}\,(\hat{\nu}) \qquad \mathcal{R}\left[r' \mapsto (\hat{\mathcal{K}}'', \mathsf{Max}\left(n', n''\right))\right]}{(\mathcal{R}, \mathcal{C}), \hat{\mathtt{I}} \overset{(r', \hat{t})}{\rightsquigarrow} \left(\mathcal{R}', \mathcal{C}\right), \hat{\mathtt{I}}}$$

(a) The reference semantics for COPS synchronisation and programs

```
1  on_receive(repl,k,v,id,deps) {
2      for (k',id') in deps { wait until (_,id',_) ∈ repl.kv(k')
           ; }
3
4      list_isnert(repl.kv[k],(v,id,deps));
5      (remote_local_time ++ replid) = id;
6      repl.local_time = max(remote_local_time, repl.local_time)
           ;
7  }
```

(b) The reference implementation for COPS synchronisation

Fig. 13: COPS synchronisation and programs

Synchronisation Between Replicas Replicas broadcast synchronisation messages for every new version that directly comes from client. Since versions are ordered and messages are assumed to be eventually delivered, this guarantees COPS satisfies eventual consistency, that is, all replicas eventually have the same state. Replica maintains queue for all new versions to be broadcasted. Once a replica receives a message with a new version $\hat{\nu}$, it accepts the version only if all the

versions that $\hat{\nu}$ depends on exist in the replica. Otherwise, the replica waits the condition to be satisfied. This is captured by Fig. 13b and COPSSync rule in Fig. 13a models the synchronisation.

Definition 43 (COPS Traces). *The set of* COPS traces COPSTraces $\ni \zeta$ *is defined by* $\text{COPSTraces} \stackrel{def}{=} \left\{ \hat{\Gamma}_0, \hat{\mathtt{P}} \twoheadrightarrow {}^*\Theta, \hat{\mathtt{P}}' \;\middle|\; \hat{\Gamma}_0 \in \text{COPSConfs}_0 \wedge \hat{\mathtt{P}}, \hat{\mathtt{P}}' \in \text{COPSProgs} \right\}$. *Two traces* ζ, ζ' *are equivalent, written* $\zeta \simeq \zeta'$*, if an only if, the last configurations are equal,* $\zeta \simeq \zeta' \stackrel{def}{\Leftrightarrow} \mathsf{LastConf}(\zeta) = \mathsf{LastConf}(\zeta')$.

Note that a COPS trace has no unfinished read operation. Two COPS traces are equivalent if and only if the last configurations match.

Normal Traces and Extended Traces A normal COPS trace is a trace that: (1) single-write transactions agree on the order over their the transaction identifiers (Eq. (8.4)); (2) the second phase of multiple-read transactions cannot be interleaved (Eq. (8.5)).

Definition 44 (Normal COPS traces). *A COPS trace* $\zeta \in$ COPSTraces *is in the* normal form *or a* normal COPS trace*, written* $\mathtt{COPSNormalTrace}(\zeta)$*, if and only if, for any clients* cl, cl'*, replicas* r, r'*, keys* k, k' *values* v, v'*, transaction identifiers* $\hat{t}, \hat{t}'$*, contexts* $\hat{u}, \hat{u}'$ *and labels* ι, ι'*,*

$$\begin{aligned} \zeta = \cdots \overset{\iota}{\rightsquigarrow} \cdots \overset{\iota'}{\rightsquigarrow} \cdots \wedge \iota &= \left(cl, r, (\mathtt{w}, k, v), \hat{t}, \hat{u}\right) \\ \wedge \iota' &= \left(cl', r', (\mathtt{w}, k', v'), \hat{t}', \hat{u}'\right) \Rightarrow \hat{t} \sqsubseteq \hat{t}', \end{aligned} \tag{8.4}$$

$$\begin{aligned} \zeta = \cdots \overset{\iota}{\rightsquigarrow} \_ \overset{\iota'}{\rightsquigarrow} \cdots \wedge \iota &= \left(cl, r, (\mathtt{w}, k, v), \hat{t}, \hat{u}, \ddagger\right) \\ \Rightarrow \iota' &= \left(cl, r, (\mathtt{r}, k', v'), \hat{t}', \hat{u}', \ddagger\right) \vee \iota' = (cl, r, \hat{u}, \ddagger). \end{aligned} \tag{8.5}$$

The normal COPS trace is the key for proving that COPS satisfies causal consistency against our semantics. In COPSWrite rule, the new version is inserted into the list of the key via COPSInsert function that ensures the order of the new list. However, by Eq. (8.4), it is enough to append the new version and it guarantees to preserve the order (Prop. 24). The second property, Eq. (8.5), ensures that any multiple-read transaction can be treated as an atomic step: only the versions that are staged in the second phase (COPSRefetch) will be returned to the client, thus it is enough to ensure no interleaving of the second phase. Every COPS trace can be transfer by swapping steps into an equivalent normal COPS trace.

Theorem 12 (Equivalent normal COPS traces). *Given a COPS trace* $\zeta \in$ COPSTraces*, there is an equivalent normal COPS trace* ζ^**, that is,* $\zeta \simeq \zeta^* \wedge \mathtt{COPSNormalTrace}(\zeta^*)$.

Proof. let $\zeta^* = \zeta$ initially. We perform the following equivalent transformation until ζ^* is a normal COPS trace. First, we move every re-fetch operation (in the second phase) to the right, that is, delaying the re-fetch, until the operation is followed by the return of the transaction or other re-fetch operation from the same transaction. Second, we move the out-of-order write operation to the left.

(1) **Right mover: second phase of reads**. Take the first read return step, $(cl, r, \hat{u}, \ddagger)$, with a re-fetch operation, $\left(cl, r, (\mathtt{r}, k, v), \hat{t}, \hat{u}', \ddagger\right)$, that is interleaved by other client, that is,

$$\zeta^* = \zeta' \overset{}{\rightsquigarrow} \_ \overset{\left(cl,r,(\mathtt{r},k,v),\hat{t},\hat{u}',\ddagger\right)}{\rightsquigarrow} \_ \overset{\iota}{\rightsquigarrow} \zeta'' \overset{(cl,r,\hat{u},\ddagger)}{\rightsquigarrow} \zeta''' \wedge \iota \neq \left(cl, r, (\mathtt{r}, k', v'), \hat{t}', \hat{u}'', \ddagger\right) \tag{8.6}$$

for three trace segments $\zeta', \zeta'', \zeta'''$. The ι can be moved left:

$$\left(\zeta' \overset{}{\rightsquigarrow} \_ \overset{\iota}{\rightsquigarrow} \_ \overset{\left(cl,r,(\mathtt{r},k,v),\hat{t},\hat{u}',\ddagger\right)}{\rightsquigarrow} \zeta'' \overset{(cl,r,\hat{u},\ddagger)}{\rightsquigarrow} \zeta'''.\right) \simeq \zeta^* \tag{8.7}$$

If the label ι is not a write operation, it is easy to see that ι and the re-fetch operation can be swapped. If the label ι is a write operation, it must not interfere the re-fetch operation. This is because any re-fetch operations can only read versions written before the lower-bound computation step. The full proof of the right mover is given in Theorem 13 on 105.
Assign the new trace to ζ^* and go back to item (1). There are finite number of steps in a trace and for each iteration, a re-fetch label moves closer to its read return label, thus item (1) must terminate. The final trace ζ^* satisfies Eq. (8.5).

(2) **Left mover: out-of-order writes**. Given a trace ζ^* satisfying Eq. (8.5), take two out-of-order write operations with the shortest distance in between, $\left(cl, r, (\mathtt{w}, k, v), \hat{t}, \hat{u}\right)$ and $\left(cl', r', (\mathtt{w}, k', v'), \hat{t}', \hat{u}'\right)$, and the operation immediately before the latter write, ι, that is,

$$\zeta^* = \zeta' \overset{}{\rightsquigarrow} \_ \overset{\left(cl,r,(\mathtt{w},k,v),\hat{t},\hat{u}\right)}{\rightsquigarrow} \zeta'' \overset{\iota}{\rightsquigarrow} \_ \overset{\left(cl',r',(\mathtt{w},k',v'),\hat{t}',\hat{u}'\right)}{\rightsquigarrow} \zeta''' \wedge \hat{t}' \sqsubseteq \hat{t},$$

where all the free variables are universally quantified. Note that there is no other out-of-order in the trace segment ζ''. Let consider the step ι:

1) if the step is not a read or write operation, the step can be delayed, that is, the out-of-order write operation can be moved to the left;
2) if the step is a write operation, by Prop. 22 it must come from a client that differs from cl', thus it is safe to move the out-of-order write operation to the left; and
3) if the step is a read operation, it is trivial that the read cannot depend on the out-of-order write operation, thus it is safe to move the out-of-order write operation to the left.

The full proof of the left mover is given in Theorem 14 on 106.
Continue to apply the left mover until the two write operations are in order:

$$\left(\zeta' \overset{}{\rightsquigarrow} \_ \overset{\left(cl',r',(\mathtt{w},k',v'),\hat{t}',\hat{u}'\right)}{\rightsquigarrow} \_ \overset{\left(cl,r,(\mathtt{w},k,v),\hat{t},\hat{u}\right)}{\rightsquigarrow} \zeta''^* \overset{\iota}{\rightsquigarrow} \zeta'''\right) \simeq \zeta^*$$

for some new trace segment ζ''^* that contains the same operations as in ζ'', however, the states must incorporate the effect of $\left(cl', r', (\mathtt{w}, k', v'), \hat{t}', \hat{u}'\right)$.

Assign the new trace to ζ^* and go back to item (2). For each iteration, two out-of-order write operations move closer to each other and eventually swap position; there are finite number of steps in a trace, thus finite number of out-of-order write operations. This means item (2) must terminate. The final trace ζ^* satisfies Eq. (8.4).

After items (1) and (2), the trace ζ^* is in the normal form as $\texttt{COPSNormalTrace}\left(\zeta^*\right)$.

Theorem 13 (Right mover: re-fetch operations). *Assume a COPS trace* $\zeta \in \text{COPSTraces}$ *that contains a re-fetch operation,* $\left(cl, r, (\mathtt{r}, k, v), \hat{t}, \hat{u}, \ddagger\right)$*, that is interleaved by other operation,* ι*:*

$$\zeta = \zeta' \overset{}{\rightsquigarrow} (\mathcal{R}, \mathcal{C}), \hat{\mathtt{I}} \overset{\left(cl,r,(\mathtt{r},k,v),\hat{t},\hat{u},\ddagger\right)}{\rightsquigarrow} (\mathcal{R}', \mathcal{C}'), \hat{\mathtt{I}}' \overset{\iota}{\rightsquigarrow} (\mathcal{R}'', \mathcal{C}''), \hat{\mathtt{I}}'' \overset{\iota'}{\rightsquigarrow} \zeta'',$$

and

$$\iota \neq \left(cl, r, (\mathtt{r}, k', v'), \hat{t}', \hat{u}', \ddagger\right) \wedge \iota \neq (cl, r, \hat{u}, \ddagger)$$

where all other free variables are universally quantified. Then the re-fetch operation can be moved to the right, that is,

$$\left(\zeta' \overset{}{\rightsquigarrow} (\mathcal{R}, \mathcal{C}), \hat{\mathtt{I}} \overset{\iota}{\rightsquigarrow} \_ \overset{\left(cl,r,(\mathtt{r},k,v),\hat{t},\hat{u},\ddagger\right)}{\rightsquigarrow} (\mathcal{R}'', \mathcal{C}''), \hat{\mathtt{I}}'' \overset{\iota'}{\rightsquigarrow} \zeta''\right) \simeq \zeta. \tag{8.8}$$

Proof. By COPSRefetch in Fig. 12a, any read from the second phase does not change the store nor the context, thus

$$\zeta = \zeta' \overset{}{\rightsquigarrow} (\mathcal{R}, \mathcal{C}), \hat{\mathtt{I}} \overset{\left(cl,r,(\mathtt{r},k,v),\hat{t},\hat{u},\ddagger\right)}{\rightsquigarrow} (\mathcal{R}, \mathcal{C}'), \hat{\mathtt{I}}' \overset{\iota}{\rightsquigarrow} (\mathcal{R}'', \mathcal{C}''), \hat{\mathtt{I}}'' \overset{\iota'}{\rightsquigarrow} \zeta''.$$

We perform case analysis on the label ι.

(1) **Case** $\left(cl', r', (\mathtt{w}, k', v'), \hat{t}', \hat{u}'\right)$. We immediately know $cl \neq cl'$. If $r \neq r'$, Eq. (8.8) trivially holds. Consider $r = r'$. Let $\hat{\mathcal{K}} = \mathcal{R}(r)$ and $\hat{\mathcal{K}}'' = \mathcal{R}''(r)$. It is easy to see that $\hat{\mathcal{K}} \sqsubseteq \hat{\mathcal{K}}''$, thus by Lemma 12, we have

$$\zeta' \overset{}{\rightsquigarrow} (\mathcal{R}, \mathcal{C}), \hat{\mathtt{I}} \overset{\iota}{\rightsquigarrow} (\mathcal{R}'', \mathcal{C}''), \hat{\mathtt{I}}^* \overset{\left(cl,r,(\mathtt{r},k,v),\hat{t},\hat{u},\ddagger\right)}{\rightsquigarrow} (\mathcal{R}'', \mathcal{C}''), \hat{\mathtt{I}}'' \overset{\iota'}{\rightsquigarrow} \zeta''.$$

for some program $\hat{\mathtt{I}}^*$; this implies Eq. (8.8).

(2) **Cases** $\left(cl', r', (\mathtt{r}, k', v'), \hat{t}', \hat{u}', \dagger\right)$, $\left(cl', r', (\mathtt{r}, k', v'), \hat{t}', \hat{u}', \dagger\right)$, $\left(cl', r', (\mathtt{r}, k', v'), \hat{t}', \hat{u}', \ddagger\right)$, $\left(cl', r', \bullet\right)$ and $\left(cl', r', \dagger\right)$. First it is easy to see that $cl \neq cl'$; and these steps do not change the states of the database nor the contexts. Therefore Eq. (8.8) holds for these steps.

(3) **Case** $\left(cl', r', \hat{u}, \ddagger\right)$. By COPSFinishRead in Fig. 12a, it is easy to see that $cl \neq cl'$; and this step does not change the states of the database. Because $cl \neq cl'$ the new context for cl' will not affect the cl, thus

$$\zeta' \overset{}{\rightsquigarrow} (\mathcal{R}, \mathcal{C}), \hat{\mathtt{I}} \overset{\iota}{\rightsquigarrow} (\mathcal{R}, \mathcal{C}''), \hat{\mathtt{I}}^* \overset{\left(cl',r',\hat{u},\ddagger\right)}{\rightsquigarrow} (\mathcal{R}'', \mathcal{C}''), \hat{\mathtt{I}}'' \overset{\iota'}{\rightsquigarrow} \zeta'',$$

which implies Eq. (8.8).

(4) **Case** $\left(r', \hat{t}\right)$. Let $\hat{\mathcal{K}} = \mathcal{R}\left(r\right)$ and $\hat{\mathcal{K}}'' = \mathcal{R}''(r)$. By COPSSync in Fig. 13a, It is easy to see that $\hat{\mathcal{K}} \sqsubseteq \hat{\mathcal{K}}''$, $\mathcal{C}' = \mathcal{C}''$ and $\hat{\mathtt{I}}' = \hat{\mathtt{I}}''$; thus by Lemma 12 and the fact that COPSSync does not rely on any context nor program, we have

$$\zeta' \xrightarrow{\iota} \left(\mathcal{R}, \mathcal{C}\right), \hat{\mathtt{I}} \xrightarrow{\iota} \left(\mathcal{R}'', \mathcal{C}\right), \hat{\mathtt{I}} \xrightarrow{\left(cl, r, (\mathtt{r}, k, v), \hat{t}, \hat{u}, \ddagger\right)} \left(\mathcal{R}'', \mathcal{C}''\right), \hat{\mathtt{I}}'' \xrightarrow{\iota'} \zeta'',$$

which implies Eq. (8.8).

Both replicas and clients are monotonic in that: (1) every replica contains increasingly more versions and the local time must tick forward; and (2) similarly, every client context contains increasingly more versions. This is a useful property that allows us to swap out-of-order write operations.

Proposition 22 (Monotonicity of COPS replica and client). *For any COPS databases $\mathcal{R}, \mathcal{R}'$, client context environments $\mathcal{C}, \mathcal{C}'$, programs $\hat{\mathtt{I}}, \hat{\mathtt{I}}'$ and label ι,*

$$\begin{aligned} &\left(\mathcal{R}, \mathcal{C}\right), \hat{\mathtt{I}} \xrightarrow{\iota}{}^{*} \left(\mathcal{R}', \mathcal{C}'\right), \hat{\mathtt{I}}' \\ &\Rightarrow \forall r \in \mathsf{Dom}\left(\mathcal{R}\right). \mathcal{R}\left(r\right)_{|0} \sqsubseteq \mathcal{R}'(r)_{|0} \wedge \mathcal{R}\left(r\right)_{|1} \leq \mathcal{R}'(r)_{|1} \\ &\qquad\qquad \wedge \forall cl \in \mathsf{Dom}\left(\mathcal{C}\right). \mathcal{C}\left(cl\right)_{|0} \subseteq \mathcal{C}'(cl)_{|0}. \end{aligned} \tag{8.9}$$

Proof. We prove by induction on the length of the trace n.
Base Case $n = 0$. Eq. (8.9) is trivially true as $\mathcal{R} = \mathcal{R}'$ and $\mathcal{C} = \mathit{copsctxenv}'$.
Base Case $n > 0$. Assume $\left(\mathcal{R}, \mathcal{C}\right), \hat{\mathtt{I}} \xrightarrow{\iota}{}^{(n-1)} \left(\mathcal{R}'', \mathcal{C}''\right), \hat{\mathtt{I}}'' \xrightarrow{\iota} \left(\mathcal{R}', \mathcal{C}'\right), \hat{\mathtt{I}}'$. By I.H., we know $\mathcal{R}\left(r\right)_{|0} \sqsubseteq \mathcal{R}'(r)_{|0}$ and $\mathcal{R}\left(r\right)_{|1} \leq \mathcal{R}'(r)_{|1}$ for $r \in \mathsf{Dom}\left(\mathcal{R}\right)$. Consider the label ι. If $\iota = \left(cl, r, (\mathtt{w}, k, v), \hat{t}, \hat{u}\right)$, by COPSWrite, we have $\mathcal{R}' = \mathcal{R}''\left[r \mapsto \left(\hat{\mathcal{K}}^*, \mathcal{R}''(r)_{|1} + 1\right)\right]$, for $\hat{\mathcal{K}}^* = \mathsf{COPSInsert}\left(\mathcal{R}''(r)_{|0}, k, \hat{\nu}\right)$ for a version $\hat{\nu}$ and, thus Eq. (8.9). If $\iota = \left(r, \hat{t}\right)$, by COPSSync we have $\mathcal{R}' = \mathcal{R}''\left[r \mapsto \left(\hat{\mathcal{K}}^*, m\right)\right]$ such that $m = \mathsf{Max}\left(\mathcal{R}''(r)_{|1}, m^*\right)$ where $\hat{t} = \hat{t}_{cl^*}^{(m^*, r^*, 0)}$, and $\hat{\mathcal{K}}' = \mathsf{COPSInsert}\left(\mathcal{R}''(r)_{|0}, k, \hat{\nu}\right)$ for a version $\hat{\nu}$, and, thus Eq. (8.9). For the rest cases, they do not change any COPS store and local time; by I.H., Eq. (8.9) holds.

Theorem 14 (Left mover: out-of-order write). *Assume a COPS trace $\zeta \in$ COPSTraces such that*

$$\zeta = \zeta' \xrightarrow{\iota} \left(\mathcal{R}, \mathcal{C}\right), \hat{\mathtt{I}} \xrightarrow{\iota} \left(\mathcal{R}', \mathcal{C}'\right), \hat{\mathtt{I}}' \xrightarrow{\left(cl, r, (\mathtt{w}, k, v), \hat{t}, \hat{u}\right)} \left(\mathcal{R}'', \mathcal{C}''\right), \hat{\mathtt{I}}'' \xrightarrow{\iota'} \zeta''$$

for a write operation $(\mathtt{w}, k, v)$ from a client cl with context $\hat{u}$ and a replica r. If the step, ι, before the write operation satisfies:

$$\begin{aligned} &\left(\iota = \left(cl', r', (\mathtt{w}, k', v'), \hat{t}', \hat{u}'\right) \wedge \left(r \neq r'\right)\right) \\ &\vee \left(\iota = \left(cl', r', (\mathtt{r}, k', v'), \hat{t}', \hat{u}', \dagger\right) \wedge \left(r = r' \wedge k' = k \Rightarrow \hat{t} \sqsubseteq \hat{t}'\right)\right) \\ &\vee \iota = \left(cl', r', (\mathtt{r}, k', v'), \hat{t}', \hat{u}', \ddagger\right) \vee \iota = \left(cl', r', \bullet\right) \\ &\vee \iota = \left(cl', r', \dagger\right) \vee \iota = \left(cl', r', \hat{u}', \ddagger\right) \vee \iota = \left(r', \hat{t}'\right), \end{aligned} \tag{8.10}$$

then the write operation can be moved left, that is,

$$\left(\zeta' \overset{\iota}{\rightsquigarrow} (\mathcal{R},\mathcal{C}), \hat{\mathtt{I}} \overset{(cl,r,(\mathtt{w},k,v),\hat{t},\hat{u})}{\rightsquigarrow} \_ \overset{\iota}{\rightsquigarrow} (\mathcal{R}'',\mathcal{C}''), \hat{\mathtt{I}}'' \overset{\iota'}{\rightsquigarrow} \zeta''\right) \simeq \zeta. \tag{8.11}$$

Proof. Note that ι are all annotated with replica r'. If $r \neq r'$ by COPSCLIENT in Fig. 13a Eq. (8.11) trivially holds. Now consider $r = r'$. We prove Eq. (8.11) by case analysis on label ι.

(1) **Case** $\left(cl', r, (\mathtt{r}, k', v'), \hat{t}', \hat{u}', \dagger\right)$. For this case, cl and cl' must be two distinct clients. If $k \neq k'$, Eq. (8.11) trivially holds. Consider $k = k'$ and $\hat{t} \sqsubseteq \hat{t}'$ Let $\hat{\nu} = (v, \hat{t}, \hat{u})$ and $\hat{\nu}' = (v', \hat{t}', \hat{u}')$. Let $\hat{\mathcal{K}} = \mathcal{R}(r)$ and $\hat{\mathcal{K}}'' = \mathcal{R}''(r)$. By COPSOPTREAD in Fig. 12a, $\hat{\mathcal{K}} = \mathcal{R}(r) = \mathcal{R}'(r)$, and $\hat{\nu}' = \hat{\mathcal{K}}\left(k, \left|\hat{\mathcal{K}}(k)\right| - 1\right)$. By COPSWRITE in Fig. 11a, we know $\hat{\mathcal{K}}'' = \mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k, \hat{\nu}'\right)$; since $\hat{t} \sqsubseteq \hat{t}'$, the version $\hat{\nu}$ is inserted in the middle of $\hat{\mathcal{K}}(k)$. Therefore we know that the last version of $\hat{\mathcal{K}}''(k)$ is still $\hat{\nu}'$, that is, $\hat{\nu}' = \hat{\mathcal{K}}''(k, \left|\hat{\mathcal{K}}''(k)\right| - 1)$. This means that the write operation does not interfere with the previous optimistic read, thus Eq. (8.11) holds.

(2) **Case** $\left(cl', r, (\mathtt{r}, k', v'), \hat{t}', \hat{u}', \dagger\right)$. For this case, cl and cl' must be two distinct clients. Let $\hat{\nu} = (v, \hat{t}, \hat{u})$ and $\hat{\nu}' = (v', \hat{t}', \hat{u}')$. Let $\hat{\mathcal{K}} = \mathcal{R}(r)$ and $\hat{\mathcal{K}}'' = \mathcal{R}''(r)$. By COPSREFETCH in Fig. 12a, $\hat{\mathcal{K}} = \mathcal{R}(r) = \mathcal{R}'(r)$; and by COPSWRITE in Fig. 11a, $\hat{\mathcal{K}}'' = \mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k', \hat{\nu}'\right)$. It is easy to see that $\hat{\mathcal{K}} \sqsubseteq \hat{\mathcal{K}}''$ defined in Lemma 12. By Lemma 12 we know that Eq. (8.11) holds.

(3) **Cases** $\left(cl', r, \bullet\right)$, $\left(cl', r, \dagger\right)$ and $\left(cl', r, \hat{u}', \ddagger\right)$. Given the rules presented in Fig. 12a, these steps are local to the client cl' without any interaction to the key-value store; thus Eq. (8.11) holds.

(4) **Case** $\left(r, \hat{t}'\right)$. A new version $\hat{\nu}'$ indexed by $\hat{t}'$ arrives to the replica r. Let $\hat{\nu} = (v, \hat{t}, \hat{u})$. Let $\hat{\mathcal{K}} = \mathcal{R}(r)$, $\hat{\mathcal{K}}' = \mathcal{R}'(r)$ and $\hat{\mathcal{K}}'' = \mathcal{R}''(r)$. By COPSSYNC in Fig. 13a, we know $\hat{\mathcal{K}}' = \mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k', \hat{\nu}'\right)$, $\hat{\mathcal{K}}'' = \mathsf{COPSInsert}\left(\hat{\mathcal{K}}', k, \hat{\nu}\right)$ and $\hat{t} \neq \hat{t}'$. By the definition of $\mathsf{COPSInsert}$, there exists $\hat{\mathcal{K}}^*$ such that

$$\hat{\mathcal{K}}^* = \mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k, \hat{\nu}\right) \wedge \hat{\mathcal{K}}'' = \mathsf{COPSInsert}\left(\hat{\mathcal{K}}^*, k, \hat{\nu}'\right),$$

and therefore, a new cops database $\mathcal{R}^*$ such that

$$\mathcal{R}^* = \mathcal{R}\left[r \mapsto \hat{\mathcal{K}}^*\right] \wedge \mathcal{R}'' = \mathcal{R}^*\left[r \mapsto \hat{\mathcal{K}}''\right].$$

The rule COPSSYNC does not depend nor change the context environment, thus we have the proof for Eq. (8.11).

Lemma 12 (Re-fetching version on a larger COPS store). *Given two COPS stores $\hat{\mathcal{K}}, \hat{\mathcal{K}}'$ such $\hat{\mathcal{K}} \sqsubseteq \hat{\mathcal{K}}'$, then*

$$\left(\hat{\mathcal{K}}, \hat{u}, n\right), \mathtt{read}(K) : (\hat{\mathcal{V}}, \hat{D}, \hat{\mathcal{V}}') \overset{(cl,r,(\mathtt{r},k_i,v_i),\hat{t}_i,\hat{d}_i,\ddagger)}{\rightsquigarrow} \left(\hat{\mathcal{K}}, \hat{u}, n\right), \mathtt{read}(K) : (\hat{\mathcal{V}}, \hat{D}, \hat{\mathcal{V}}'')$$

$$\Rightarrow \left(\hat{\mathcal{K}}', \hat{u}, n\right), \mathtt{read}(K) : (\hat{\mathcal{V}}, \hat{D}, \hat{\mathcal{V}}') \overset{(cl,r,(\mathtt{r},k_i,v_i),\hat{t}_i,\hat{d}_i,\ddagger)}{\rightsquigarrow} \left(\hat{\mathcal{K}}', \hat{u}, n\right), \mathtt{read}(K) : (\hat{\mathcal{V}}, \hat{D}, \hat{\mathcal{V}}'')$$

Proof. Depending on $\hat{\mathcal{V}}_{|i}$, we have two possible cases.

(1) **Case** $\mathsf{WriterOf}\left(\hat{\mathcal{V}}_{|i}\right) \sqsubseteq \hat{t}_i$. For this case, there exists an version $\hat{\mathcal{K}}\left(k_i, m\right) = \left(v_i, \hat{t}_i, \hat{d}_i\right)$ for some index m. Given a new store $\hat{\mathcal{K}}'$ such that $\hat{\mathcal{K}} \sqsubseteq \hat{\mathcal{K}}'$, this version must be also included in $\hat{\mathcal{K}}'$, thus $\hat{\mathcal{K}}'(k_i, m') = \left(v_i, \hat{t}_i, \hat{d}_i\right)$ for some m'; therefore we have the proof.

(2) **Case** $\mathsf{WriterOf}\left(\hat{\mathcal{V}}_{|i}\right) = \hat{t}_i$. For this case, the state of $\hat{\mathcal{K}}$ is irrelevant.

In our kv-store semantics (§ 3), we assign a unique identifier for every transition (CAtomicTrans), but in COPS there is explicit transaction identifier. We encode the COPS version identifier as the transaction identifier for the single-write transaction who committed the version. We then annotate multiple-read transactions with transaction identifiers that preserves the session order.

Definition 45 (Extended COPS traces). A client local time environments *is defined by* $\hat{\mathcal{C}} : cl \to$ COPSTxIDs*. Given a normal COPS trace* $\zeta \in$ COPSTraces*, the extended COPS trace for* ζ *is defined by*

$$\mathsf{COPSToExt}\left(\hat{\Gamma}_0, \hat{\mathcal{C}}\right) \stackrel{def}{=} \hat{\Gamma}_0,$$

$$\mathsf{COPSToExt}\left(\left(\left(\mathcal{R}, \mathcal{C}\right), \hat{\mathtt{I}} \overset{\iota}{\rightsquigarrow} \zeta\right), \hat{\mathcal{C}}\right)$$

$$\stackrel{def}{=} \begin{cases} \left(\mathcal{R}, \mathcal{C}\right), \hat{\mathtt{I}} \overset{\iota}{\rightsquigarrow} \mathsf{COPSToExt}\left(\zeta, \hat{\mathcal{C}}\left[cl \mapsto \hat{t}_{cl}^{(n,r,1)}\right]\right) & \text{if } \iota = \left(cl, r, \left(\mathtt{w}, k, v\right), \hat{t}_{cl}^{(n,r,0)}, \hat{u}\right), \\ \left(\mathcal{R}, \mathcal{C}\right), \hat{\mathtt{I}} \overset{\iota, \hat{\mathcal{C}}(cl)}{\rightsquigarrow} \mathsf{COPSToExt}\left(\zeta, \hat{\mathcal{C}}\right) & \text{if } \iota = \left(cl, r, \left(\mathtt{r}, k, v\right), \hat{t}, \hat{u}, \ddagger\right), \\ \left(\mathcal{R}, \mathcal{C}\right), \hat{\mathtt{I}} \overset{\iota, \hat{\mathcal{C}}(cl)}{\rightsquigarrow} \mathsf{COPSToExt}\left(\zeta, \hat{\mathcal{C}}\left[cl \mapsto \hat{t}_{cl}^{(n,r,m+1)}\right]\right) & \text{if } \iota = \left(cl, r, \hat{u}, \ddagger\right) \wedge \hat{\mathcal{C}}\left(r\right) = \hat{t}_{cl}^{(n,r,m)}, \\ \left(\mathcal{R}, \mathcal{C}\right), \hat{\mathtt{I}} \overset{\iota}{\rightsquigarrow} \mathsf{COPSToExt}\left(\zeta, \hat{\mathcal{C}}\right) & \text{otherwise.} \end{cases}$$

Then the set of extended COPS traces, ExtCOPSTraces $\ni \hat{\zeta}$*, is defined by defined by*

$$\textsc{ExtCOPSTraces} \stackrel{def}{=} \left\{ \mathsf{COPSToExt}\left(\zeta, \hat{\mathcal{C}}\right) \;\middle|\; \begin{array}{l} \zeta \in \textsc{COPSTraces} \\ \wedge\, \exists \mathcal{C} \in \textsc{COPSCtxEnvs}.\, (\_, \mathcal{C}) = \zeta_{|0} \\ \Rightarrow \forall cl \in \mathsf{Dom}\left(\mathcal{C}\right).\, \forall r \in \textsc{COPSReps}. \\ \qquad \mathcal{C}\left(cl\right) = (\_, r) \Rightarrow \hat{\mathcal{C}}\left(cl\right) = \hat{t}_{cl}^{(0,r,1)} \end{array} \right\}.$$

We track the next available multiple-read transaction identifier for every client, that is, $\hat{\mathcal{C}}$. Recall that each COPS transaction identifier $\hat{t}_{cl}^{(n,r,m)}$ is annotated with a extra counter m, namely the read counter. In the COPS semantics, the read counter is always set to be ZERO. Here, we assign transaction identifiers with non-zero read counters to multiple-read transactions: for each multiple-read transaction, we annotate the re-fetch operations and the read return operation with the next available identifier in $\hat{\mathcal{C}}$. To preserve the session order, we update $\hat{\mathcal{C}}$ for the client cl, if cl commits a new transaction. By construction, the transaction identifiers that are assigned to multiple-read transactions must be unique.

Proposition 23 (Fresh multiple-read transaction identifiers). *Given an extended COPS trace* $\hat{\zeta} \in \textsc{ExtCOPSTraces}$*, the annotated multiple-read transaction identifiers must be fresh with respect the key being read, that is,*

$$\begin{aligned}&\hat{\zeta} = \hat{\zeta}' \xrightarrow{} (\mathcal{R}, \mathcal{C}), \hat{\mathtt{I}} \xrightarrow{\left(cl, r, (\mathtt{r}, k, v), \hat{t}, \hat{d}, \ddagger\right), \hat{t}'} \hat{\zeta}'' \\ &\wedge \forall r^* \in \textsc{COPSReps}.\, \forall cl^* \in \textsc{ClientID}.\, \forall v^* \in \textsc{Value}.\, \forall \hat{d}^* \in \textsc{COPSDeps}. \\ &\qquad \hat{\zeta}'' = \cdots \xrightarrow{\left(cl^*, r^*, (\mathtt{r}, k, v^*), \hat{t}^*, \hat{d}^*, \ddagger\right), \hat{t}''} \cdots \Rightarrow \hat{t}' \neq \hat{t}''\end{aligned}$$

Proof. Suppose a replica r^*, a client cl^*, a value v^* and dependency set $\hat{d}^*$ such that

$$\hat{\zeta}'' = \cdots \xrightarrow{\left(cl^*, r^*, (\mathtt{r}, k, v^*), \hat{t}^*, \hat{d}^*, \ddagger\right), \hat{t}''} \cdots.$$

If $r \neq r^*$ or $cl \neq cl^*$, by the definition of ExtCOPSTraces (especially COPSToExt), it is trivial that $\hat{t} \neq \hat{t}''$. Consider $r = r^*$ or $cl = cl^*$. Because a client must provide a unique set of key K when calling `read` by the rules in Fig. 12a, it means that

$$\hat{\zeta}'' = \cdots \xrightarrow{\left(cl, r, \hat{d}', \ddagger\right), \hat{t}'} \cdots \xrightarrow{\left(cl^*, r^*, (\mathtt{r}, k, v^*), \hat{t}^*, \hat{d}^*, \ddagger\right), \hat{t}''} \cdots.$$

For this case, by the definition of ExtCOPSTraces (especially COPSToExt), it must be the case that $\hat{t} \neq \hat{t}''$.

In our kv-store semantics, new versions are appended to the end, however, in COPS semantics, new versions are inserted into the lists to preserve the version order within the lists. As explained before, write operations in a normal COPS trace, henceforth in an extended COPS trace, must be in order. Thus, it is safe to *append* any write operation to the end of the list of the key, instead of inserting into the list.

Proposition 24 (Appending write operation). *Given an extended COPS trace* $\hat{\zeta}$*, for every write operation,* $\iota = \left(cl, r, (\mathtt{w}, k, v), \hat{t}, \hat{u}\right)$*, the version identifier,* $\hat{t}$*, is strictly greater than all version identifiers included in all replicas, which means that*

$$\begin{aligned}&\hat{\zeta} = \hat{\zeta}' \xrightarrow{} (\mathcal{R}, \mathcal{C}), \hat{\mathtt{I}} \xrightarrow{\left(cl, r, (\mathtt{w}, k, v), \hat{t}, \hat{u}\right)} (\mathcal{R}', \mathcal{C}'), \hat{\mathtt{I}}' \xrightarrow{} \cdots \\ &\wedge \forall r' \in \textsc{COPSReps}.\, \forall \hat{\mathcal{K}}' \in \textsc{COPSKVSs}.\, \forall k' \in \textsc{Keys}.\, \forall i' \in \mathbb{N}. \\ &\quad \mathcal{R}(r') = (\hat{\mathcal{K}}', \_) \wedge 0 \leq i < \hat{\mathcal{K}}(k') \Rightarrow \mathsf{WriterOf}\left(\hat{\mathcal{K}}'(k', i')\right) \sqsubset \hat{t}\end{aligned}$$

Proof. Suppose a replica r' with key-value store $\hat{\mathcal{K}}'$ $(\mathcal{R}(r') = (\hat{\mathcal{K}}', \_))$, a key k' and an index i' such that $0 \leq i < \hat{\mathcal{K}}(k')$. Let the $\hat{t}' = \mathsf{WriterOf}\left(\hat{\mathcal{K}}(k', i')\right)$. There must exist a step with label ι' with some client cl' in the trace $\hat{\zeta}'$ that committed this version:

$$\hat{\zeta}' = \cdots \xrightarrow{\iota'} \cdots \wedge \iota' = \left(cl, r, (\mathtt{w}, k', v'), \hat{t}', \hat{d}\right)$$

for some value v' and dependent set $\hat{d}$. Since an extended trace $\hat{\zeta}$ is also in its normal form, that is, $\texttt{COPSNormalTrace}\left(\hat{\zeta}\right)$, and the new kv-store $\mathcal{R}'(r')_{|0}$ must be a well-formed COPS key-value store, therefore $\hat{t}' \sqsubset \hat{t}$.

Trace Refinement In this section, we show how to encode a COPS machine state to our centralised kv-store, and then client contexts to views on the kv-store.

For each extended COPS trace, the final configuration corresponds to a kv-store by replying all the transactions in order on the initial kv-store. Note that, the final configuration itself contains enough information for all the write transactions, however, we need the trace to recover the annotated identifies for multiple-read transactions.

Definition 46 (Centralised COPS kv-store). *Given an extended COPS trace $\hat{\zeta} \in$ ExtCOPSTraces the centralised kv-store induced by the trace, written $\mathsf{COPSToKVS}\left(\hat{\zeta}\right)$, is defined in Fig. 14.*

Recall the properties of extended COPS traces:

(1) multiple-read transactions are annotated with unique identifiers (Prop. 23); and
(2) it is safe to append new versions (Prop. 24).

This mean that the function $\mathsf{COPSToKVS}$ is well-defined and more importantly returns a well-formed kv-store.

Proposition 25 (Well-formed COPS kv-store). *Given an extended COPS trace $\hat{\zeta}$, the kv-store $\mathcal{K} = \mathsf{COPSToKVS}\left(\hat{\zeta}\right)$ is a well-formed COPS key-value store, that is, $\texttt{WfKvs}\left(\mathcal{K}\right)$, and it contains and only contains versions in the last configuration $(\mathcal{R}, \mathcal{C}) = \mathsf{LastConf}\left(\hat{\zeta}\right)$:*

$$\forall k \in \textsc{Keys}.\, \forall i \in \mathbb{N}.\, \forall v \in \textsc{Value}.\, \forall \hat{t} \in \text{COPSTxIDs}.\, \mathcal{K}\left(k, i\right) = (v, \hat{t}, \_) \Rightarrow \exists i' \in \mathbb{N}.\, \exists r \in \text{COPSReps}.\, \mathcal{R}\left(r\right)\left(k, i'\right) = (v, \hat{t}, \_), \quad (8.12)$$

$$\forall k \in \textsc{Keys}.\, \forall i \in \mathbb{N}.\, \forall v \in \textsc{Value}.\, \forall \hat{t} \in \text{COPSTxIDs}.\, \forall r \in \text{COPSReps}.\, \mathcal{R}\left(r\right)\left(k, i'\right) = (v, \hat{t}, \_) \Rightarrow \exists i' \in \mathbb{N}.\, \mathcal{K}\left(k, i'\right) = (v, \hat{t}, \_). \quad (8.13)$$

Proof. We prove this by induction on the length of trace $\hat{\zeta}$.

(1) **Cases** $\hat{\zeta} = \hat{\Gamma}_0$. By definition of $\mathsf{COPSToKVS}$, the kv-store $\mathcal{K}_0 = \mathsf{COPSToKVS}\left(\hat{\Gamma}_0\right)$ is trivially well-formed. By the definition of $\hat{\Gamma}_0$, Eqs. (8.12) and (8.13) hold.
(2) **Cases** $\hat{\zeta} = \hat{\zeta}' \overset{\iota}{\rightsquigarrow} (\mathcal{R}, \mathcal{C}), \hat{\mathtt{I}}$. Let $\mathcal{K}' = \mathsf{COPSToKVS}\left(\hat{\zeta}'\right)$; by I.H., the kv-store $\mathcal{K}'$ is a well-formed kv-store and satisfies Eqs. (8.12) and (8.13). Consider label ι.

$$\mathsf{COPSToKVS}\left(\hat{\Gamma}_0, \hat{\mathtt{P}}\right) \stackrel{def}{=} \mathcal{K}_0,$$

$$\mathsf{COPSToKVS}\left(\zeta \overset{\iota}{\rightarrow} (\mathcal{R}, \mathcal{C}), \hat{\mathtt{I}}\right) \stackrel{def}{=} \mathtt{let}\ \mathcal{K} = \mathsf{COPSToKVS}(\zeta)\ \mathtt{in}$$
$$\begin{cases} \mathcal{K}\left[k \mapsto \mathcal{K}(k) :: \left[\left(v, \hat{t}_{cl}^{(n,r,0)}, \emptyset\right)\right]\right] & \text{if } \iota = \left(cl, r, (\mathtt{w}, k, v), \hat{t}_{cl}^{(n,r,0)}, \hat{u}\right), \\ \mathcal{K}\left[k \mapsto \mathcal{K}(k)\left[i \mapsto \left(v, \hat{t}, T \uplus \hat{t}_{cl}^{(n,r,m)}\right)\right]\right] & \text{if } \iota = \left(\left(cl, r, (\mathtt{r}, k, v), \hat{t}, \hat{u}, \ddagger\right), \hat{t}_{cl}^{(n,r,m)}\right), \wedge\mathcal{K}(k)_{|i} = (v, \hat{t}, T), \\ \mathcal{K} & \text{otherwise.} \end{cases}$$

$$\mathsf{COPSToKVTrace}\left(\hat{\Gamma}_0, \hat{\mathtt{P}}\right) \stackrel{def}{=} \mathtt{let}\ s_0 = \lambda \mathtt{x} \in .\, 0\ \mathtt{in}\left(\mathsf{COPSToKVS}\left(\hat{\Gamma}_0, \hat{\mathtt{P}}\right), \mathsf{COPSViews}\left(\hat{\Gamma}_0, \hat{\mathtt{P}}\right), \lambda cl \in \mathsf{Dom}\left(\hat{\mathtt{P}}\right).\, s_0\right), \mathsf{COPSToKVProg}\left(\hat{\mathtt{P}}\right)$$

$$\mathsf{COPSToKVTrace}\left(\hat{\zeta} \overset{\iota}{\rightarrow} \hat{\zeta}' \overset{\iota'}{\rightsquigarrow} (\mathcal{R}, \mathcal{C}), \hat{\mathtt{I}}\right) \stackrel{def}{=} \mathtt{let}\ (\mathcal{K}, \mathcal{U}, \mathcal{E}), \mathtt{P} = \mathsf{LastConf}\left(\mathsf{COPSToKVTrace}\left(\hat{\zeta}\right)\right),$$
$$\mathcal{K}' = \mathsf{COPSToKVS}\left(\hat{\zeta} \overset{\iota}{\rightarrow} \hat{\zeta}' \overset{\iota'}{\rightsquigarrow} (\mathcal{R}, \mathcal{C}), \hat{\mathtt{P}}\right),$$
$$\mathcal{U}' = \mathsf{COPSViews}(\mathcal{R}, \mathcal{C}),\ \mathtt{and}\ \mathtt{P}' = \mathsf{COPSToKVProg}\left(\hat{\mathtt{I}}\right)\ \mathtt{In}$$
$$\begin{cases} \mathsf{COPSToKVTrace}\left(\hat{\zeta}\right) \xrightarrow{(cl, \mathcal{U}(cl), \mathcal{F})}_{\top} (\mathcal{K}', \mathcal{U}', \mathcal{E}), \mathtt{P}' & \text{if } \dagger, \\ \mathsf{COPSToKVTrace}\left(\hat{\zeta}\right) \xrightarrow{(cl, \mathcal{U}'(cl), \mathcal{F})}_{\top} (\mathcal{K}', \mathcal{U}', \mathcal{E}'), \mathtt{P}' & \text{if } \ddagger, \\ \mathsf{COPSToKVTrace}\left(\hat{\zeta}\right) & \text{other cases } \iota = \iota' \text{ and } \hat{\zeta} = \emptyset. \end{cases}$$

$$\dagger \equiv \iota = \iota' = \left(cl, r, (\mathtt{w}, k, v), \hat{t}, \hat{u}\right) \wedge \mathcal{F} = \{(\mathtt{w}, k, v)\}$$

$$\ddagger \equiv \iota = \left(cl, r, (\mathtt{r}, k_0, v_0), \hat{t}_0, \hat{u}_0, \ddagger\right), \hat{t} \wedge \hat{\zeta}' = \_ \overset{\left(cl, r, (\mathtt{r}, k_1, v_1), \hat{t}_1, \hat{u}_1, \ddagger\right), \hat{t}}{\rightsquigarrow} \cdots \overset{\left(cl, r, (\mathtt{r}, k_n, v_n), \hat{t}_n, \hat{u}_n, \ddagger\right), \hat{t}}{\rightsquigarrow} \_$$
$$\wedge\, \iota' = \left(cl, r, \hat{u}', \ddagger\right), \hat{t} \wedge \mathcal{F} = \{(\mathtt{r}, k_0, v_0), \cdots, (\mathtt{r}, k_n, v_n)\} \wedge \mathcal{E}' = \mathcal{E}\left[\mathtt{x}_0 \mapsto v_0\right] \cdots \left[\mathtt{x}_n \mapsto v_n\right]$$

Fig. 14: Definitions of COPSToKVS and COPSToKVTrace

1) **Cases** $\iota = \left(cl, r, (\mathtt{w}, k, v), \hat{t}_{cl}^{(n,r,0)}, \hat{u}\right)$. By the definition of $\mathsf{COPSToKVS}$, the new kv-store $\mathcal{K} = \mathsf{COPSToKVS}\left(\hat{\zeta}\right)$ is given by $\mathcal{K} = \mathcal{K}'\left[k \mapsto \mathcal{K}'(k) :: \left[\left(v, \hat{t}_{cl}^{(n,r,0)}, \emptyset\right)\right]\right]$. By COPSWrite, the transaction identifier for the new version must be fresh, that is, $\hat{t} \notin \mathcal{K}$, which implies $\mathtt{WfKvs}\,(\mathcal{K})$. Let $(\mathcal{R}, \mathcal{C}) = \mathsf{LastConf}\left(\hat{\zeta}\right)$ and $(\hat{\mathcal{K}}, \_) = \mathcal{R}\,(r)$; The new version must be included in $\hat{\mathcal{K}}$: $\hat{\mathcal{K}}\left(k, \left|\hat{\mathcal{K}}\,(k)\right| - 1\right) = (v, \hat{t}_{cl}^{(n,r,0)}, \hat{d})$ for some dependent set $\hat{d}$, which means Eqs. (8.12) and (8.13).
2) **Cases** $\iota = \left(\left(cl, r, (\mathtt{w}, k, v), \hat{t}, \hat{u}\right), \hat{t}_{cl}^{(n,r,m)}\right)$. By the definition of $\mathsf{COPSToKVS}$, the new kv-store $\mathcal{K} = \mathsf{COPSToKVS}\left(\hat{\zeta}\right)$ is given by $\mathcal{K} = \mathcal{K}'\left[k \mapsto \mathcal{K}'(k)\left[i \mapsto \left(v, \hat{t}, T \cup \hat{t}_{cl}^{(n,r,m)}\right)\right]\right]$ and $\mathcal{K}'(k)_{|i} = (v, \hat{t}, T)$ for some i, v, T. By Prop. 23, the transaction identifier $\hat{t}_{cl}^{(n,r,m)}$ must be fresh with respect to the key k, that is $\hat{t}_{cl}^{(n,r,m)} \notin \mathcal{K}'(k, i')$ for all index i' in range; this implies $\mathtt{WfKvs}\,(\mathcal{K})$. There is no new version; therefore the new kv-store $\mathcal{K}$ satisfies Eqs. (8.12) and (8.13).
3) **Cases** other ι. By the definition of $\mathsf{COPSToKVS}$, the new kv-store $\mathcal{K} = \mathsf{COPSToKVS}\left(\hat{\zeta}\right)$; and by I.H. it is a well-formed kv-store and satisfies Eqs. (8.12) and (8.13).

Given the kv-store encoding the final configuration of an extended COPS trace, the final context for every client can be encoded to a view on the kv-store: the view contains all versions ν that are included in the context and versions that ν depends on, directly or indirectly.

Definition 47 (COPS context views). *Assume an extended COPS trace $\hat{\zeta}$. Given the last configuration $(\mathcal{R}, \mathcal{C}) = \mathsf{LastConf}\left(\hat{\zeta}\right)$ and the kv-store $\mathcal{K} = \mathsf{COPSToKVS}\left(\hat{\zeta}\right)$, the view environment induced by the client context $\mathcal{C}$, written $\mathsf{COPSViews}\,(\mathcal{R}, \mathcal{C})$, is defined by*

$$\mathsf{COPSViews}\,(\mathcal{R}, \mathcal{C})\,(cl) \stackrel{def}{=} \lambda k \in \textsc{Keys}.\left\{ i \;\middle|\; \begin{array}{l} \exists k, k' \in \textsc{Keys}.\, \exists \hat{t}, \hat{t}' \in \text{COPSTxIDs}. \\ (k, \hat{t}) \in \mathcal{C}\,(cl) \wedge (k', \hat{t}') \xrightarrow{\mathsf{DEP}_{\mathcal{R}}{}^*} (k, \hat{t}) \\ \qquad\qquad \wedge\, \hat{t}' = \mathsf{WriterOf}\,(\mathcal{K}\,(k', i)) \end{array} \right\} \cup \{0\},$$

where the COPS dependency relation $\mathsf{DEP}_{\mathcal{R}}$ *is defined by*

$$\mathsf{DEP}_{\mathcal{R}} \stackrel{def}{=} \bigcup_{\mathcal{K} \in \mathsf{Image}(\mathcal{R})} \left\{ \left((k, \hat{t}), (k', \hat{t}')\right) \;\middle|\; \begin{array}{l} \exists i.\, \hat{t}' = \mathsf{WriterOf}\,(\mathcal{K}\,(k', i)) \\ \qquad \wedge\, (k, \hat{t}) \in \mathsf{DepSetOf}\,(\mathcal{K}\,(k', i)) \end{array} \right\}$$

The relation $\mathsf{DEP}_{\mathcal{R}}$ on a COPS database $\mathcal{R}$ denotes dependency relations between versions: if $\left((k, \hat{t}), (k', \hat{t}')\right) \in \mathsf{DEP}_{\mathcal{R}}$, then $(k, \hat{t})$ is included in the dependent set of the newer version $\hat{t}'$.

Since all versions have unique transactions, it is easy to see that the view induced by the COPS client context is well-formed.

Proposition 26 (Well-formed COPS context views). *Assume an extended COPS trace* $\hat{\zeta}$*. Let* $\mathcal{K} = \mathsf{COPSToKVS}\left(\hat{\zeta}\right)$ *and* $\mathcal{U} = \mathsf{COPSViews}\left(\mathsf{LastConf}\left(\hat{\zeta}\right)\right)$*. Then every view in* $\mathcal{U}$ *is a well-formed view on the kv-store* $\mathcal{K}$*, that is,* $\mathcal{U}(cl) \in \textsc{Views}(\mathcal{K})$ *for all clients* $cl \in \mathsf{Dom}(\mathcal{U})$*.*

Proof. Assume a client cl and the view $u = \mathcal{U}(cl)$ for the client cl. By definition of $\mathsf{COPSViews}$, Eq. (well-formed) are trivially true. Consider keys k, k' and indexes i, i' such that $i \in u(k)$ and $\mathsf{WriterOf}(\mathcal{K}(k,i)) = \mathsf{WriterOf}(\mathcal{K}(k',i'))$. By Eq. (8.2) $k = k'$ and $i = i'$ therefore Eq. (atomic) holds.

Last, it is easy to convert the COPS syntax, namely `put` and `read` APIs, to explicit transactional syntax.

Definition 48 (COPS Atomic Transactions). *Given a COPS command* $\hat{\mathtt{C}} \in \textsc{COPSCmds}$*, the (kv-store) command* $\mathsf{COPSToKVCmd}\left(\hat{\mathtt{C}}\right) : \textsc{Commands}$ *induced by* $\hat{\mathtt{C}}$ *is defined by*

$$\mathsf{COPSToKVCmd}\left(\mathtt{put}\left(k,v\right)\right) \stackrel{def}{=} \left[[k] := v\right],$$

$$\mathsf{COPSToKVCmd}\left(\mathtt{read}\left([k_0,\cdots,k_n]\right)\right) \stackrel{def}{=} \left[\mathtt{x}_0 := [k_0]\,;\cdots;\mathtt{x}_n := [k_n]\right],$$

$$\mathsf{COPSToKVCmd}\left(\hat{\mathtt{R}}\,;\hat{\mathtt{R}}'\right) \stackrel{def}{=} \mathsf{COPSToKVCmd}\left(\hat{\mathtt{R}}\right);\mathsf{COPSToKVCmd}\left(\hat{\mathtt{R}}'\right).$$

The (kv-store) program induced by a COPS program is defined by

$$\mathsf{COPSToKVProg}\left(\hat{\mathtt{P}}\right) = \lambda cl \in \mathsf{Dom}\left(\hat{\mathtt{P}}\right).\,\mathsf{COPSToKVCmd}\left(\hat{\mathtt{P}}(cl)\right).$$

We now encode the COPS trace $\hat{\zeta}$ into a $\mathtt{ET}_\top$-trace τ. In the encode, we throw away all the internal steps, optimistic reads and synchronisations between replicas, since those steps are opaque to clients.

Definition 49 (COPS Kv-store Traces). *Given an extended COPS trace* $\hat{\zeta} \in \textsc{ExtCOPSTraces}$ *the kv-store traces converted from* ζ*, written* $\mathsf{COPSToKVTrace}(\zeta)$*, is defined in Fig. 14.*

For any single-write transaction for a client in the COPS trace $\hat{\zeta}$, we simulate in $\mathtt{ET}_\top$-trace by a step with a singleton fingerprint of the write operation from the same client. For any multiple-read transaction, we ignore operations in the first phase, and combine all the steps from the second phase into an atomic transaction in $\mathtt{ET}_\top$-trace. The fingerprint of this transaction contains all the re-fetch operations. We now prove the $\mathtt{ET}_\top$-trace τ is a $\mathtt{CC}$-trace.

Theorem 15 (COPS causal consistency). *Given an extended COPS trace* $\hat{\zeta} \in \textsc{ExtCOPSTraces}$ *the kv-store traces,* $\eta = \mathsf{COPSToKVTrace}(\zeta)$*, can be obtained under* $\mathtt{CC}$*.*

Proof. Recall the definition of $\texttt{CC}$ that for kv-stores $\mathcal{K}, \mathcal{K}'$, views u, u' and fingerprint $\mathcal{F}$,

$$u \sqsubseteq u' \tag{8.14}$$

$$\forall t \in \mathcal{K}' \setminus \mathcal{K}.\, \forall k \in \textsc{Keys}.\, \forall i \in \mathbb{N}.\, \mathsf{WriterOf}\left(\mathcal{K}'(k,i)\right) \xrightarrow{\mathsf{SO}^?} t \Rightarrow i \in u'(k) \tag{8.15}$$

$$\texttt{PreClosed}\left(\mathcal{K}, u, \mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO}\right) \tag{8.16}$$

We first show that every step in the trace η satisfies Eq. (8.14); then we show that every step *preserves* Eqs. (8.15) and (8.16) for all views included in the view environment. We now prove by induction on the length of the trace η.

(1) **Base Case** $\eta = \left(\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}\right), \texttt{P}$. By the definition of $\mathcal{K}_0$ and $\mathcal{U}_0$, It is trivial that Eqs. (8.15) and (8.16) hold.

(2) **Inductive Case** $\eta = \eta' \xrightarrow{\iota}_{\top} \left(\mathcal{K}, \mathcal{U}, \mathcal{E}\right), \texttt{P}$ for $\eta = \mathsf{COPSToKVTrace}\left(\hat{\zeta}\right)$ and $\texttt{CC}$-trace $\eta' = \mathsf{COPSToKVTrace}\left(\hat{\zeta}'\right)$. Consider the label ι.

1) **Case** $\iota = \left(cl, u, \{(\texttt{w}, k, v)\}\right)$. By definition of $\mathsf{COPSToKVTrace}$, traces $\hat{\zeta}, \hat{\zeta}'$ satisfy the following property

$$\begin{aligned} u = \mathsf{COPSViews}\left(\hat{\zeta}'\right)(cl) \wedge \hat{\zeta} = \hat{\zeta}' \xrightarrow{\left(cl, r, (\texttt{w}, k, v), \hat{t}_{cl}^{(n,r,0)}, \hat{d}\right)} \left(\mathcal{R}, \mathcal{C}\right), \hat{\texttt{I}} \\ \wedge \mathsf{LastConf}\left(\hat{\zeta}'\right) = \left(\mathcal{R}', \mathcal{C}'\right) \\ \wedge \mathcal{C}' = \mathcal{C}\left[cl \mapsto \left(\mathcal{C}\left(cl\right)_{|0} \cup \left\{\left(k, \hat{t}_{cl}^{(n,r,0)}\right)\right\}, r\right)\right] \end{aligned} \tag{8.17}$$

Let $u' = \mathsf{COPSViews}\left(\hat{\zeta}\right)(cl) = \mathcal{U}\left(cl\right)$ be the view after udpate. By Eq. (8.17) and the definition of $\mathsf{COPSViews}$, the view must incresse as $u \sqsubseteq u'$ and thus Eq. (8.14); also the new view u' must contains the new version written by $\hat{t}_{cl}^{(n,r,0)}$ and thus Eq. (8.15). By definition of $\mathsf{COPSViews}$ and Prop. 27, it follows Eq. (8.16).

2) **Case** $\iota = \left(cl, u, \{(\texttt{r}, k_0, v_0), \cdots, (\texttt{r}, k_n, v_n)\}\right)$. By definition of $\mathsf{COPSToKVTrace}$, traces $\hat{\zeta}, \hat{\zeta}'$ satisfy the following property

$$\begin{aligned} u = \mathcal{U}\left(cl\right) \\ \wedge \hat{\zeta} = \hat{\zeta}' \xrightarrow{\left(cl, r, (\texttt{r}, k_0, v_0), \hat{t}_0, \hat{u}_0, \ddagger\right), \hat{t}} \cdots \xrightarrow{\left(cl, r, (\texttt{r}, k_n, v_n), \hat{t}_n, \hat{u}_n, \ddagger\right), \hat{t}} \cdots \xrightarrow{\left(cl, r, \hat{u}', \ddagger\right), \hat{t}} \left(\mathcal{R}, \mathcal{C}\right), \hat{\texttt{I}} \\ \wedge \mathsf{LastConf}\left(\hat{\zeta}'\right) = \left(\mathcal{R}', \mathcal{C}'\right) \wedge \mathcal{C}' = \mathcal{C}\left[cl \mapsto \left(\mathcal{C}\left(cl\right)_{|0} \cup \hat{u}', r\right)\right] \end{aligned} \tag{8.18}$$

By Eq. (8.18) and the definition of $\mathsf{COPSViews}$, the view before and after read are the same view u and thus Eq. (8.14); since there is no new version and view cannot lost any version, thus Eq. (8.15) holds.

By rule COPSFinishRead, it follows $(k_i, \hat{t}_i) \in \hat{u}'$ for index i such that $0 \leq i \leq n$. By Prop. 27,

$$\mathsf{WriterOf}\,(\mathcal{K}\,(k,m)) \xrightarrow{(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO})^+} \hat{t}_i \Rightarrow j \in u\,(k)\,, \tag{8.19}$$

then by rules COPSLowerBound and COPSRefetch,

$$\mathsf{WriterOf}\,(\mathcal{K}\,(k,m)) \xrightarrow{(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO})^+} \hat{t}_i \wedge \exists j.\, k = k_j \Rightarrow \mathsf{WriterOf}\,(\mathcal{K}\,(k,m)) \sqsubseteq \hat{t}_j. \tag{8.20}$$

Combining Eqs. (8.19) and (8.20), the closure property Eq. (8.16) holds.

Proposition 27 (COPS dependency relation to CC relation). *Given an extended COPS trace $\hat{\zeta} \in$ ExtCOPSTraces, the kv-store traces $\eta = \mathsf{COPSToKVTrace}\,(\zeta)$, let $\mathcal{K}$ be the finial kv-store such that $(\mathcal{K}, \_, \_) = \mathsf{LastConf}\,(\eta)$; let $\mathcal{R}$ be the finial state of COPS database such that $(\mathcal{R}, \_) = \mathsf{LastConf}\,(\zeta)$. Given two versions $\mathcal{K}\,(k,i)$ written $\hat{t}$ and $\mathcal{K}\,(k',i')$ written by $\hat{t}'$, if $\hat{t} \xrightarrow{(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO})^+} \hat{t}'$ then $(k, \hat{t}) \xrightarrow{\mathsf{DEP}_{\mathcal{R}}{}^+} (k', \hat{t}')$.*

Proof. By the hypothesis, both transactions are single-write transactions which means $(t, t') \notin \mathsf{WR}_{\mathcal{K}}$. For any transaction in COPS, it either is a read-only transaction or a single-write transaction. Recall that SO is traisitive. This means that $\hat{t} \xrightarrow{(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO})^+} \hat{t}'$ if and only if $\hat{t} \xrightarrow{(\mathsf{WR}_{\mathcal{K}}{}^? ; \mathsf{SO})^+} \hat{t}'$ and thus it is sufficient to prove the following result

$$\hat{t} \xrightarrow{\mathsf{WR}_{\mathcal{K}}{}^? ; \mathsf{SO}} \hat{t}' \Rightarrow (k, \hat{t}) \xrightarrow{\mathsf{DEP}_{\mathcal{R}}{}^+} (k', \hat{t}'). \tag{8.21}$$

Case $\hat{t} \xrightarrow{\mathsf{SO}} \hat{t}'$. Assume that $\hat{t}, \hat{t}'$ are from client cl that interacts with replica r. For this case, transaction $\hat{t}$ must commit before $\hat{t}'$, that is

$$\zeta = \cdots \xrightarrow{(cl, r, (\mathtt{w}, k, v), \hat{t}, \hat{u})} \cdots \xrightarrow{(cl, r, (\mathtt{w}, k', v'), \hat{t}', \hat{u}')} \cdots.$$

By rule COPSWrite and COPS invariant in Prop. 22, it follows $(k, \hat{t}) \in \hat{u}'$ which implies Eq. (8.21).

Case $\hat{t} \xrightarrow{\mathsf{WR}_{\mathcal{K}}} \hat{t}'' \xrightarrow{\mathsf{SO}} \hat{t}'$ for a read-only transaction $\hat{t}''$. Assume that $\hat{t}'', \hat{t}'$ are from client cl that interacts with replica r. For this case, we know

$$\zeta = \cdots \xrightarrow{(cl, r, (\mathtt{r}, k, v), \hat{t}, \hat{d}, \ddagger), \hat{t}''} \cdots \xrightarrow{(cl, r, \hat{u}, \ddagger), \hat{t}''} \cdots \xrightarrow{(cl, r, (\mathtt{w}, k', v'), \hat{t}', \hat{u}')} \cdots.$$

By rules COPSRefetch and COPSFinishRead, $(k, \hat{t}) \in \hat{u}$ and then by COPS invariant in Prop. 22, it follows $(k, \hat{t}) \in \hat{u}'$ which implies Eq. (8.21).

H.2 Clock-SI

Code

Structure Clock-SI is a partitioned distributed NoSQL database, which means each server, also called shard, contains part of keys and does not overlap with any other servers. Clock-SI implements snapshot isolation. To achieve that, each shard tracks the physical time. Note that times between shards do not match, but there is a upper bound of the difference.

```
1 Shard :: ID -> ( clockTime )
```

Code 1.1: Shard

A key maintains a list of values and their versions. A version is the time when such value is committed.

```
1 VersionNo :: Time
2 KV :: Keys -> List( Val, VersionNo )
3 (each key is asscoaited with a shard)
```

Code 1.2: Key-value store

The idea behind Clock-SI is that a client starts a transaction in a shard, and the shard is responsible for fetching value from other shards if keys are not stored in the local shard. During execution, a transaction tracks the write set.

```
1 WS :: Key -> Val
```

Code 1.3: Write set

At the end, the transaction commits all the update in the write set, and the local shard acts as coordinator to update keys either locally or remotely. To commit a transaction, Clock-SI use two-phase commits protocol. A transaction has four states:

- `active`, the transaction is still running;
- `prepared`, shards receive the update requests from the coordinator;
- `committing`, shards receive the update confirmations from the coordinator;
- `committed`, the transaction commits successfully.

To implement SI, a transaction also tracks its snapshot time so it knows which version should be fetched. Also a transaction tracks the prepared and committing times, which are used to postpone other transactions' reads if those transactions' snapshots time are greater.

```
1 State :: { active, prepared, committing, committed }
2 Trans :: ( state, snapTime, prepareTime, commitTime, ws )
```

Code 1.4: Transaction runtime

Start Transaction Clock-SI proposes two versions, with or without session order. Here we verify the one with session order. To start a transaction, the client contacts a shard and provides the previous committing time. The shard will return a snapshot time, which is greater than the committing time provided, for the new transaction. Note that client might connects to a different shard from last time, which means that the shard might have to wait until the shard local time is greater than the committing time.

```
startTransaction( Trans t, Time ts )
    wait until ts < getClockTime();
    t.snapshotTime = getClockTime();
    t.state = active;
```

Code 1.5: Transaction runtime

From this point, such transaction will always interact with the shard and the shard will act as coordinator if necessary.

Read A transaction `t` might read within the transaction if the key has been updated by the same transaction before, that is, read from the write set `ws`. A transaction `t` might read from the original shard if the key store in the shard, but it has to wait until any other transactions `t'` commit successfully who are supposed to commit before the current transaction's snapshot time, i.e. `t'` are in `prepared` or `committing` stage and the corresponding time is less the `t` snapshot time.

```
Read( Trans t, key k )
    if ( k in t.ws ) return ws(k);
    if ( k is updated by t' and t'.state = committing
                and t.snapshotTime > t'.committingTime)
        wait until t'.state == committed;
    if ( k is updated by t'
                    and t'.state = prepared and t.
                        snapshotTime > t'.preparedTime
                    and t.snapshotTime > t'.committingTime )
        wait until t'.state == committed;
    return K(k,i), where i is the latest version before t.
        snapshotTime;
```

Code 1.6: Read from original shard

If the key is not stored in the original shard, the original shard sends a read request to the shard containing the key. Because of time difference, the remote shard's time might before the snapshot time of the transaction. In this case, the shard wait until the time catches up.

```
On read k request from a remote transaction t
    wait until t.snapshotTime < getClockTime()
    return read(t,k);
```

Code 1.7: Read from original shard

Commit Write Set If all the keys in the write set are hosted in the original shard that the transaction first connected, the write set only needs to commit local.

```
localCommit( Trans t )
    if noConcurrentWrite(t) {
        t.state = committing;
```

```
4           t.commitTime = getClockTime();
5           log t.commitTime;
6           log t.ws;
7           t.state = committed;
8       }
```

Code 1.8: Local Commit

To commit local, it first checks, by `noConcurrentWrite(t)`, if there is any transaction `t'` that writes to the same key as the transaction new transaction `t`, and the transaction `t'` commit after the snapshot of `t`. Since writing database needs time, it sets the transaction state to `committing` and log the `commitTime`, before the updating really happens. During `committing` state, other transactions will be pending, if they want to read the keys being updated. Last, the state of transaction is set to `committed`.

To commit to several shards, Clock-SI uses two-phase protocol.

```
1  distributedCommit( Trans t )
2      for p in t.updatedPartitions { send ‘‘prepare t’’ to p; }
3      wait receiving ‘‘t prepared’’ from all participants,
           store into prep;
4      t.state = committing;
5      t.commitTime = max(prep);
6      log t.commitTime;
7      t.state = committed;
8      for p in t.updatedPartitions { send ‘‘commit t’’ to p; }
9
10 On receiving ‘‘prepare t’’
11     if noConcurrentWrite(t) {
12         log t.ws and t.coordinator ID
13         t.state = prepared;
14         t.prepareTime = getClockTime();
15         send ‘‘t prepared’’ to t.coordinator
16     }
17
18 On receiving ‘‘commit t’’
19     log t.commitTime
20     t.state = committed
```

Code 1.9: Distributed Commit

The original shard, who acts as the coordinator, sends `''prepare t''` to shards that will be updated. Any shard receiving `''prepare t''` checks, similarly, if there is any transaction write to the same key committing after the snapshot time. If the check passes, the shard logs the write set and the coordinator shard ID, set the state to `prepared`, and sends the local time to the coordinator. Once the coordinator receives all the `prepared` messages, it starts the second phase by setting the state to `committing`. Then the coordinator picks the largest time from all the `prepared` messages as the commit time for the new transaction. Since the write set has been logged in the first phase, so here it can immediately

set the state to be `committed`. Last, the coordinator needs to send `commit t` to other shards so they will log the commit time and set the state to `committed`. Note that participants have different view for the new transaction from the coordinator, but it guarantees eventually they all updated to `committed` with the same commit time.

Verification

Structure We model the database use key-value store from Def. 1, yet here it is necessary to satisfy the well-formed conditions. Transaction identifier t^n_{cl} are labelled with the committing time n. Sometime we also write t^c_{cl} or omit the client label, i.e. t^n and t^c.

Database is partitioned into several *shards*. A shard $r \in \textsc{Shards}$ contains some keys which are disjointed from keys in other shards. The $\mathsf{shardOf}(k)$ denotes the shard where the key k locates.

Shards and clients are associated with clock times, $c \in \textsc{ClockTimes} \triangleq \mathbb{N}$, which represent the current times of shards and clients. We use notation $\mathcal{C} \in (\textsc{Shards} \cup \textsc{ClientID}) \xrightarrow{fin} \textsc{ClockTimes}$.

We will use notation $\left[\mathtt{T}\right]$ to denote the static code of a transaction, and $\left[\mathtt{T}\right]^{\mathcal{F}}_{c}$ for the runtime of a transaction, where c is the snapshot time and $\mathcal{F}$ is the read-write set. Note that in the model, we only distinguishes `active` and `committed` state, since the `prepared` and `committing` are only for better performance.

Start Transaction To start a transaction, it picks a random shard r as the coordinator, reads the local time $\mathcal{C}(r)$ as the snapshot, and sets the initial read-write set to be an empty set. Also the client time is updated to the snapshot time. For technical reason, we also update the shard time to avoid time collision to other transaction about to commit. Note that in real life, all the operations running in a shard take many time cycles, so it is impossible to have time collision.

$$\textsc{StartTrans}\ \frac{c < \mathcal{C}(r) \qquad \mathcal{C}' = \mathcal{C}[r \mapsto \mathcal{C}(r)+1] \quad \texttt{---> simulate time elapses}}{cl \vdash \mathcal{K}, c, \mathcal{C}, s, \left[\mathtt{T}\right] \xrightarrow{\mathcal{C}(r), c, \emptyset, \bot} \mathcal{K}, \mathcal{C}(r), \mathcal{C}', s, \left[\mathtt{T}\right]^{\emptyset}_{\mathcal{C}(r)}}$$

Read The clock-SI protocol includes some codes related to performance which does not affect the correctness. Clock-SI distinguishes a local read/commit and a remote read/commit, yet it is sufficient to assume all the read and commit are "remote", while the local read and commit can be treated as self communication. Similarly we assume a transaction always commits to several shards.

```
1  On receive ‘‘read(t,k)’’ {
2      if ( k in t.ws ) return ws(k);
3
4      asssert( t.snapshotTime < getClockTime() )
5      for t’ that writes to k:
6          if(t.snapshotTime > t’.preparedTime
```

```
7                       || t.snapshotTime > t'.committingTime)
8               asssert( t.state == committed )
9
10      return K(k,i), where i is the latest version before t.
            snapshotTime;
11  }
```

Code 1.10: simplified read

If the key exists in the write set $\mathcal{F}$, the transaction read from the write set immediately.

ReadTrans

$$\frac{k = [\![\mathtt{E}]\!]_s \qquad (\mathtt{w}, k, v) \in \mathcal{F} \quad \texttt{---> Code 1.10, line 2}}{cl \vdash \mathcal{K}, c, \mathcal{C}, s, \left[\mathtt{x} := [\mathtt{E}]\right]_c^{\mathcal{F}} \xrightarrow{cl, c, \mathcal{F} \lll (\mathtt{r}, k, v), \bot} \mathcal{K}, c, \mathcal{C}, s[\mathtt{x} \mapsto v], \left[\mathtt{skip}\right]_c^{\mathcal{F} \lll (\mathtt{r}, k, v)}}$$

Otherwise, the transaction needs to fetch the value from the shard. The first premiss says the transaction must wait until the shard local time $\mathcal{C}(\mathsf{shardOf}(k))$ is greater than the snapshot time c. If so, by the second line, the transaction fetches the latest version for key k before the snapshot time c.

ReadRemote

$$\frac{\begin{array}{c} k = [\![\mathtt{E}]\!]_s \qquad (\mathtt{w}, k, \_) \notin \mathcal{F} \qquad c < \mathcal{C}(\mathsf{shardOf}(k)) \quad \texttt{---> Code 1.10, line 4} \\ n = \max \left\{ n' \;\middle|\; \exists j.\ t^{n'} = \mathsf{w}(\mathcal{K}(k, j)) \wedge n' < c \right\} \quad \texttt{---> Code 1.10, line 10} \\ \mathcal{K}(k, i) = (v, t^n, \_) \end{array}}{cl \vdash \mathcal{K}, c, \mathcal{C}, s, \left[\mathtt{x} := [\mathtt{E}]\right]_c^{\mathcal{F}} \xrightarrow{cl, c, \mathcal{F} \lll (\mathtt{r}, k, v), \bot} \mathcal{K}, c, \mathcal{C}', s[\mathtt{x} \mapsto v], \left[\mathtt{skip}\right]_c^{\mathcal{F} \lll (\mathtt{r}, k, v)}}$$

Write Write will not go to the shard until committing time. Before it only log it in the write set.

Write

$$\frac{k = [\![\mathtt{E}_1]\!]_s \qquad v = [\![\mathtt{E}_2]\!]_s}{cl \vdash \mathcal{K}, c, \mathcal{C}, s, \left[[\mathtt{E}_1] := \mathtt{E}_2\right]_c^{\mathcal{F}} \xrightarrow{cl, c, \mathcal{F} \lll (\mathtt{w}, k, v), \bot} \mathcal{K}, c, \mathcal{C}, s, \left[\mathtt{skip}\right]_c^{\mathcal{F} \lll (\mathtt{w}, k, v)}}$$

Commit We also assume transaction always commit to several shards and the local commit is treated as self-communication.

```
1  commit( Trans t )
2      for p in t.updatedPartitions
3          send ‘‘prepare t’’ to p;
4      wait receiving ‘‘t prepared’’ from all participants,
           store into prep;
5      t.state = committing;
6      t.commitTime = max(prep);
7      log t.commitTime;
8      t.state = committed;
9      for p in t.updatedPartitions
10         send ‘‘commit t’’ to p;
11
```

```
12 On receiving ‘‘prepare t’’
13     if noConcurrentWrite(t)
14         log t.ws to t.coordinator ID
15         t.state = prepared;
16         t.prepareTime = getClockTime();
17         send ‘‘t prepared’’ to t.coordinator
18
19 On receiving ‘‘commit t’’
20     log t.commitTime
21     t.state = committed
```

Code 1.11: simplified commit

Note that Clock-SI uses two phase commit: the coordinator (the shard that the client directly connects to) distinguishes "committing" state and "committed" state, where in between the coordinator pick the committing time and log the write set, and the participants distinguishes "prepared" state and "committed" state. Such operations are for possible network partition or single shard errors, and allowed a more fine-grain implementations which do not affect the correctness, therefore it suffices to assume they are one atomic step.

Commit

$$\frac{\begin{array}{c}\forall k, i.\ (\mathtt{w}, k, \_) \in \mathcal{F} \wedge \mathsf{w}(\mathcal{K}(k,i)) < c \quad \texttt{---> Code 1.11, line 13} \\ n = \max\left(\{c' \mid \exists k.\ (\_, k, \_) \in \mathcal{F} \wedge c' = \mathcal{C}(\mathsf{shardOf}(k))\} \cup \{c\}\right) \quad \texttt{---> Code 1.11, lines 4 to 6} \\ \mathcal{K}' = \mathsf{commitKV}(\mathcal{K}, c, t^n_{cl}, \mathcal{F}) \quad \texttt{---> Code 1.11, lines 20 and 21} \\ \forall r.\ \begin{pmatrix} r \in \{\mathsf{shardOf}(k) \mid (\_, k, \_) \in \mathcal{F}\} \\ \Rightarrow \mathcal{C}'(r) = \mathcal{C}(r) + 1 \end{pmatrix} \vee (\mathcal{C}'(r) = \mathcal{C}(r)) \quad \texttt{---> simulate time elapses}\end{array}}{r, cl \vdash \mathcal{K}, c, \mathcal{C}, s, \left[\mathtt{skip}\right]^{\mathcal{F}}_{c} \xrightarrow{cl, c, \mathcal{F}, n} \mathcal{K}', n+1, \mathcal{C}', s, \mathtt{skip}}$$

To commit the new transaction, it needs to check, by the first premiss, there is no other transactions writing to the same keys after the snapshot time. If it passes, by the second line it picks the maximum time n among all participants as the commit time. The new key-value store $\mathcal{K}' = \mathsf{commitKV}(\mathcal{K}, c, t^n_{cl}, \mathcal{F})$, where

$$\begin{aligned}\mathsf{commitKV}(\mathcal{K}, c, t, \mathcal{F} \uplus \{(\mathtt{r}, k, v)\}) \triangleq\ & \mathtt{let}\ n = \max\left\{n' \;\middle|\; \exists j.\ t^{n'} = \mathsf{w}(\mathcal{K}(k,j)) \wedge n' < c\right\} \\ & \quad \mathsf{and}\ \mathcal{K}(k,i) = (v, t^n, T) \\ & \quad \mathsf{and}\ \mathcal{K}' = \mathsf{commitKV}(\mathcal{K}, c, t, \mathcal{F}) \\ & \quad \mathtt{in}\ \mathcal{K}'[k \mapsto \mathcal{K}'(k)[i \mapsto (v, t^n, T \cup \{t\})]]\end{aligned}$$

$$\begin{aligned}\mathsf{commitKV}(\mathcal{K}, c, t, \mathcal{F} \uplus \{(\mathtt{w}, k, v)\}) \triangleq\ & \mathtt{let}\ \mathcal{K}' = \mathsf{commitKV}(\mathcal{K}, c, t, \mathcal{F}) \\ & \quad \mathtt{in}\ \mathcal{K}'[k \mapsto \mathcal{K}'(k)::(v, t, \emptyset)]\end{aligned}$$

Note that commitKV is similar to update by appending the new version to the end of a list. The commitKV also updates versions read by the new transaction using the snapshot time of the transaction. Last, like StartTrans we update the client time after the commit time, i.e. $n+1$ and simulate time elapses for all shards updated.

Time Tick For technical reasoning, we have non-deterministic time elapses.

$$\textsc{TimeTick}\quad \frac{}{\mathcal{K},\mathcal{C},\mathcal{C}',\mathcal{E},\mathtt{P} \xrightarrow{r,\mathcal{C}(r)+1} \mathcal{K},\mathcal{C},\mathcal{C}'[r \mapsto \mathcal{C}(r)+1],\mathcal{E},\mathtt{P}}$$

$$\textsc{ClientStep}\quad \frac{cl \vdash \mathcal{K},\mathcal{C}(cl),\mathcal{C}',\mathcal{E}(cl),\mathtt{P}\,(cl) \xrightarrow{cl,c',\mathcal{F},c''} \mathcal{K}',c,\mathcal{C}'',s,\mathtt{C}}{\mathcal{K},\mathcal{C},\mathcal{C}',\mathcal{E},\mathtt{P} \xrightarrow{cl,c',\mathcal{F},c''} \mathcal{K},\mathcal{C}[cl \mapsto c],\mathcal{C}'',\mathcal{E}[cl \mapsto s],\mathtt{P}[cl \mapsto \mathtt{C}]}$$

Verification Clock-SI allows interleaving, yet for any clock-si trace τ there exists a equivalent trace τ' where transactions do not interleave with others (Theorem 16). Furthermore, in such trace τ', transactions are reduced in their commit order.

Theorem 16 (Normal clock-SI trace). *A clock-SI trace τ is a clock-SI normal trace if it satisfies the following: there is no interleaving of a transaction,*

$$\begin{aligned}&\forall cl, c, \mathcal{K}_i, \mathcal{C}_i, \mathcal{C}'_i, \mathcal{E}_i, \mathtt{P}_i.\\ &\qquad \tau = \cdots \xrightarrow{cl,c,\_,\bot} \mathcal{K}_i,\mathcal{C}_i,\mathcal{C}'_i,\mathcal{E}_i,\mathtt{P}_i \xrightarrow{-} \cdots\\ &\qquad\qquad \Rightarrow \exists c', \mathcal{K}_j,\mathcal{C}_j,\mathcal{C}'_j,\mathcal{E}_j,\mathtt{P}_j.\\ &\tau = \cdots \xrightarrow{cl,c,\_,\bot} \mathcal{K}_i,\mathcal{C}_i,\mathcal{C}'_i,\mathcal{E}_i,\mathtt{P}_i \xrightarrow{cl,c,\_,\bot} \_ \xrightarrow{cl,c,\_,\bot} \cdots \xrightarrow{cl,c,\_,c'} \mathcal{K}_j,\mathcal{C}_j,\mathcal{C}'_j,\mathcal{E}_j,\mathtt{P}_j\end{aligned} \tag{8.22}$$

and transactions in the trace appear in the committing order,

$$\begin{aligned}&\forall cl_i, cl_j, c_i, c_j, c'_i, cl'_j, \mathcal{F}_i, \mathcal{F}_j \mathcal{K}_i, \mathcal{K}_j, \mathcal{C}_i, \mathcal{C}_j, \mathcal{C}'_i, \mathcal{C}'_j, \mathcal{E}_i, \mathcal{E}_j, \mathtt{P}_i, \mathtt{P}_j.\\ &\tau = \cdots \xrightarrow{cl_i,c_i,\mathcal{F}_i,c'_i} \mathcal{K}_i,\mathcal{C}_i,\mathcal{C}'_i,\mathcal{E}_i,\mathtt{P}_i \xrightarrow{-} \cdots \xrightarrow{cl_j,c_j,\mathcal{F}_j,c'_j} \mathcal{K}_j,\mathcal{C}_j,\mathcal{C}'_j,\mathcal{E}_j,\mathtt{P}_j \Rightarrow c'_i < c'_j\end{aligned} \tag{8.23}$$

For any clock-SI trace τ, there exists an equivalent normal trace τ' which has the same final configuration as τ.

Proof. Given a trace τ, we first construct a trace τ' that satisfies Eq. (8.23), by swapping steps. Let take the first two transactions $t^n_{cl_i}$ and $t^m_{cl_j}$ that are out of order, i.e. $n > m$ and

$$\tau = \cdots \xrightarrow{cl_i,c_i,\mathcal{F}_i,n} \mathcal{K}_i,\mathcal{C}_i,\mathcal{C}'_i,\mathcal{E}_i,\mathtt{P}_i \xrightarrow{-} \cdots \xrightarrow{cl_j,c_j,\mathcal{F}_j,m} \mathcal{K}_j,\mathcal{C}_j,\mathcal{C}'_j,\mathcal{E}_j,\mathtt{P}_j$$

By Lemma 13, the two clients are different $cl_i \neq cl_j$ and thus two steps are unique in the trace. We will construct a trace that $t^n_{cl_i}$ commits after $t^m_{cl_j}$.

- First, it is important to prove that $t^m_{cl_j}$ does not read any version written by $t^n_{cl_i}$. By Lemma 21, the snapshot time c_j of $t^m_{cl_j}$ is less than the commit time, i.e. $c_j < m$, therefore $c_j < n$. By the READ rule, $c_j < n$ implies the transaction $t^m_{cl_j}$ never read any version written by $t^n_{cl_i}$.

– Let consider any possible time tick for those shard r that has been updated by $t^m_{cl_j}$, that is, $r = \mathsf{shardOf}(k)$ for some key k that $(\mathtt{w}, k, \_) \in \mathcal{F}_i$ and

$$\tau = \cdots \xrightarrow{cl_i, c_i, \mathcal{F}_i, n} \mathcal{K}_i, \mathcal{C}_i, \mathcal{C}'_i, \mathcal{E}_i, \mathtt{P}_i \xrightarrow{-} \cdots \xrightarrow{r,c} \_ \xrightarrow{-} \cdots \xrightarrow{cl_j, c_j, \mathcal{F}_j, m} \mathcal{K}_j, \mathcal{C}_j, \mathcal{C}'_j, \mathcal{E}_j, \mathtt{P}_j \tag{8.24}$$

Since $c_j < m < n < c$, therefore such time tick will not affect the transaction $t^m_{cl_j}$, which means it is safe to move the time tick step after the $t^m_{cl_j}$.

Now we can move the commit of $t^n_{cl_i}$ and time tick steps similar to Eq. (8.24) after the commit of $t^m_{cl_j}$,

$$\tau' = \cdots \xrightarrow{cl_j, c_j, \mathcal{F}_j, m} \mathcal{K}_j, \mathcal{C}_j, \mathcal{C}'_j, \mathcal{E}_j, \mathtt{P}_j \xrightarrow{cl_i, c_i, \mathcal{F}_i, n} \mathcal{K}_i, \mathcal{C}_i, \mathcal{C}'_i, \mathcal{E}_i, \mathtt{P}_i \xrightarrow{r,c} \cdots$$

We continually swap the out of order transaction until the newly constructed trace τ' satisfying Eq. (8.23).

Now let consider Eq. (8.22). Let take the first transaction t whose read has been interleaved by other transaction or a time tick.

– If it is a step that the transaction t read from local state,

$$\tau = \cdots \xrightarrow{cl, c, \mathcal{F} \lll (\mathtt{r}, k, v), \bot} \_ \xrightarrow{\alpha} \cdots \xrightarrow{cl, c, \mathcal{F}'', n} \cdots$$

then by ReadTrans we know $\mathcal{F} \lll (\mathtt{r}, k, v) = \mathcal{F}$, and it is safe to swap the two steps as the following

$$\tau' = \cdots \xrightarrow{\alpha} \_ \xrightarrow{cl, c, \mathcal{F} \lll (\mathtt{r}, k, v), \bot} \cdots \xrightarrow{cl, c, \mathcal{F}'', n} \cdots$$

– If it is a step that the transaction t read from remote, the step might be interleaved by a step from other transaction or time tick step.
  • if it is interleaved by the commit of other transaction $t' = t^m_{cl'}$, that is

$$\tau = \cdots \xrightarrow{cl, c, \mathcal{F}, \bot} \mathcal{K}, \mathcal{C}, \mathcal{C}', \mathcal{E}, \mathtt{P} \xrightarrow{cl', c', \mathcal{F}', m} \cdots \xrightarrow{cl, c, \mathcal{F}'', n} \cdots$$

where $cl' \neq cl$.
    ∗ if the transaction t' does not write to any key k that is read by t,

$$\forall k.\ (\mathtt{r}, k, \_) \in \mathcal{F} \Rightarrow (\mathtt{w}, k, \_) \notin \mathcal{F}'$$

In this case, it is safe to swap the two steps

$$\tau' = \cdots \xrightarrow{cl', c', \mathcal{F}', m} \_ \xrightarrow{cl, c, \mathcal{F}, \bot} \cdots \xrightarrow{cl, c, \mathcal{F}'', n} \cdots$$

    ∗ if the transaction t' write to a key k that is read by t,

$$(\mathtt{r}, k, \_) \in \mathcal{F} \wedge (\mathtt{w}, k, \_) \in \mathcal{F}'$$

Let $r = \mathsf{shardOf}(k)$. By the ReadRemote, we know the current clock time for the shard r is greater than c which is the snapshot time of t, that is, $\mathcal{C}'(r) > c$. Then by commit, the commit time of t' is picked as the maximum of the shards it touched, i.e. $m \geq \mathcal{C}'(r)$. Now by the ReadRemote and $m \geq c$, it is safe to swap the two steps since the new version of k does not affect the t.

- if it is interleaved by the read of other transaction t', that is

$$\tau = \cdots \xrightarrow{cl,c,\mathcal{F},\bot} \mathcal{K},\mathcal{C},\mathcal{C}',\mathcal{E},\mathtt{P} \xrightarrow{cl',c',\mathcal{F}',\bot} \cdots \xrightarrow{cl,c,\mathcal{F}'',n} \cdots$$

  Because reads have no side effect to any shard by READREMOTE, it is safe to swap the two steps

$$\tau' = \cdots \xrightarrow{cl',c',\mathcal{F}',\bot} \_ \xrightarrow{cl,c,\mathcal{F},\bot} \cdots \xrightarrow{cl,c,\mathcal{F}'',n} \cdots$$

- if it is interleaved by a time tick step,

$$\tau = \cdots \xrightarrow{cl,c,\mathcal{F}\ll\lhd(\mathtt{r},k,v),\bot} \mathcal{K},\mathcal{C},\mathcal{C}',\mathcal{E},\mathtt{P} \xrightarrow{r,c'} \cdots \xrightarrow{cl,c,\mathcal{F}'',n} \cdots$$

  * if the transaction t does not read from the shard r, it means for any key k,

$$\mathsf{shardOf}(k) \neq r$$

    In this case, it is safe to swap the two steps

$$\tau' = \cdots \xrightarrow{r,c'} \_ \xrightarrow{cl,c,\mathcal{F}\ll\lhd(\mathtt{r},k,v),\bot} \cdots \xrightarrow{cl,c,\mathcal{F}'',n} \cdots$$

  * if the transaction t read from the shard r, it means that there exists a key k

$$\mathsf{shardOf}(k) = r$$

    By the READREMOTE, we know the current clock time for the shard r is greater than the snapshot time of t, that is, $\mathcal{C}'(r) > c$. Then by TIMETICK, we have $c' > \mathcal{C}'(r)$. Now by the READREMOTE and $c' > c$, it is safe to swap the two steps.

Lemma 13 (Monotonic client clock time). *The clock time associated with a client monotonically increases, That is, given a step*

$$\mathcal{K},\mathcal{C},\mathcal{C}',\mathcal{E},\mathtt{P} \xrightarrow{-} \mathcal{K}',\mathcal{C}'',\mathcal{C}''',\mathcal{E}',\mathtt{P}'$$

then for any clients cl,

$$\mathcal{C}(cl) \leq \mathcal{C}(cl')$$

Proof. It suffices to only check the CLIENTSTEP rule which is the only rule updates the client clock time, especially, it is enough to check the client cl that who starts or commits a new transaction.

– COMMIT. Let c be the clock time before committing, $c = \mathcal{C}(cl)$. By the premiss of the rule, the new client time $n+1$ satisfies that,

$$n = \max\left(\{c' \mid \exists k.\ (\_,k,\_) \in \mathcal{F} \wedge c' = \mathcal{C}(\mathsf{shardOf}(k))\} \cup \{c\}\right)$$

  It means $c < (n+1)$.

– StartTrans. Let c be the clock time before taking snapshot, $c = \mathcal{C}(cl)$. By the premiss of the rule the new client time $\mathcal{C}'(r)$ for a shard r, such that $c < \mathcal{C}'(r)$.

Lemma 14 (No side effect local operation). *Any transactional operation has no side effect to the shard and key-value store,*

$$\mathcal{K}, \mathcal{C}, \mathcal{C}', \mathcal{E}, \mathtt{P} \xrightarrow{cl,c,\mathcal{F},\bot} \mathcal{K}', \mathcal{C}'', \mathcal{C}''', \mathcal{E}', \mathtt{P}' \Rightarrow \mathcal{K} = \mathcal{K}'$$

Proof. It is easy to see that StartTrans, ReadTrans, ReadRemote and Write do not change the state of key-value store.

Clock-SI also has a notion view which corresponds the snapshot time. The following definition $\mathsf{viewOf}(\mathcal{K}, c)$ extracts the view from snapshot time.

Definition 50. *Given a normal clock-SI trace τ and a transaction t_{cl}, such that*

$$\tau = \cdots \xrightarrow{cl,c,\emptyset,\bot} \cdots \xrightarrow{cl,c,\mathcal{F},\bot} \mathcal{K}, \mathcal{C}, \mathcal{C}', \mathcal{E}, \mathtt{P} \xrightarrow{cl,c,\mathcal{F},c'} \cdots$$

the initial view of the transaction is defined as the following:

$$\mathsf{viewOf}(\mathcal{K}, c) \triangleq \lambda k.\, \{i \mid \exists t^n.\ \mathsf{w}(\mathcal{K}(k,i)) = t^n \wedge n < c\}$$

Given the view $\mathsf{viewOf}(\mathcal{K}, c)$ for each transaction, we first prove that clock-si produces a well-formed key-value store (Def. 1).

Lemma 15. *Given any key-value store $\mathcal{K}$ and snapshot time c from a clock-SI trace τ,*

$$\tau = \cdots \dot{\to} \mathcal{K}, \mathcal{C}, \mathcal{C}', \mathcal{E}, \mathtt{P} \to cl, c, \mathcal{F}, c' \cdots$$

$\mathsf{viewOf}(\mathcal{K}, c)$ *and* $\mathsf{viewOf}(\mathcal{K}, c')$ *(Def. 50) produce well-formed views.*

Proof. It suffices to prove that Eq. (atomic) in Def. 2. Assume a key-value store $\mathcal{K}$ and a snapshot time c. Suppose a version i in the view $i \in \mathsf{viewOf}(\mathcal{K}, c)(k)$ for some key k. By Def. 50, the version is committed before the snapshot time, i.e. $t^n = \mathsf{w}(\mathcal{K}(k,i)) \wedge n < c$. Assume another version $t^n = \mathsf{w}(\mathcal{K}(k',j))$ for some key k' and index j. By Def. 50 we have $j \in \mathsf{viewOf}(\mathcal{K}, c)(k')$. Similarly $\mathsf{viewOf}(\mathcal{K}, c')$ is a well-formed view.

Second, given the view $\mathsf{viewOf}(\mathcal{K}, c)$ for each transaction, both $\mathsf{commitKV}$ and update produce the same result.

Lemma 16. *Given a normal clock-SI trace τ and a transaction t_{cl}, such that*

$$\tau = \cdots \xrightarrow{cl,c,\emptyset,\bot} \cdots \xrightarrow{cl,c,\mathcal{F},\bot} \mathcal{K}, \mathcal{C}, \mathcal{C}', \mathcal{E}, \mathtt{P} \xrightarrow{cl,c,\mathcal{F},c'} \cdots$$

the following holds:

$$\mathsf{commitKV}\Big(\mathcal{K}, c, t_{cl}^{c'}, \mathcal{F}\Big) = \mathsf{update}\Big(\mathcal{K}, \mathsf{viewOf}(\mathcal{K}, c), \mathcal{F}, t_{cl}^{c'}\Big)$$

Proof. We prove by induction on $\mathcal{F}$.

- Base case: $\mathcal{F} = \emptyset$. It is easy to see that

$$\mathsf{commitKV}\Big(\mathcal{K}, c, t^{c'}_{cl}, \emptyset\Big) = \mathcal{K} = \mathsf{update}\mathcal{K}, \mathsf{viewOf}(\mathcal{K}, c), \mathcal{F}, t^{c'}_{cl}$$

- Inductive case: $\mathcal{F} \uplus (\mathtt{w}, k, v)$. Because in both functions, the new version is installed at the tail of the list associated with k,

$$\begin{aligned}
&\mathsf{commitKV}\Big(\mathcal{K}, c, t^{c'}_{cl}, \mathcal{F} \uplus (\mathtt{w}, k, v)\Big) \\
&\quad = \mathsf{commitKV}\Big(\mathcal{K}, c, t^{c'}_{cl}, \mathcal{F}\Big)[k \mapsto \mathcal{K}(k)::(v, t, \emptyset)] \\
&\quad = \mathsf{update}\Big(\mathcal{K}, \mathsf{viewOf}(\mathcal{K}, c), \mathcal{F}, t^{c'}_{cl}\Big)[k \mapsto \mathcal{K}(k)::(v, t, \emptyset)] \\
&\quad = \mathsf{update}\Big(\mathcal{K}, \mathsf{viewOf}(\mathcal{K}, c), \mathcal{F} \uplus (\mathtt{w}, k, v), t^{c'}_{cl}\Big)
\end{aligned}$$

- Inductive case: $\mathcal{F} \uplus (\mathtt{r}, k, v)$. Let $\mathcal{K}(k, i)$ be the version being read. That is, the writer $t^n = \mathsf{w}(\mathcal{K}(k, i))$ is the latest transaction written to the key k before the snapshot time c,

$$n = \max\left\{n' \;\middle|\; \exists j.\ t^{n'} = \mathsf{w}(\mathcal{K}(k, j)) \wedge n' < c\right\}$$

  Let the new version $\nu = \Big(\mathsf{val}(\mathcal{K}(k, i)), \mathsf{w}(\mathcal{K}(k, i)), \mathsf{rs}(\mathcal{K}(k, i)) \uplus \Big\{t^{c'}_{cl}\Big\}\Big)$. By Lemma 15, it follows $i \in \mathsf{viewOf}(\mathcal{K}, c)(k)$, then by Lemma 17, the version is the latest one $i = \max(\mathsf{viewOf}(\mathcal{K}, c)(k))$. Therefore we have,

$$\begin{aligned}
&\mathsf{commitKV}\Big(\mathcal{K}, c, t^{c'}_{cl}, \mathcal{F} \uplus (\mathtt{r}, k, v)\Big) \\
&\quad = \mathsf{commitKV}\Big(\mathcal{K}, c, t^{c'}_{cl}, \mathcal{F}\Big)[k \mapsto \mathcal{K}(k)[i \mapsto \nu]] \\
&\quad = \mathsf{update}\Big(\mathcal{K}, \mathsf{viewOf}(\mathcal{K}, c), \mathcal{F}, t^{c'}_{cl}\Big)[k \mapsto \mathcal{K}(k)[i \mapsto \nu]] \\
&\quad = \mathsf{update}\Big(\mathcal{K}, \mathsf{viewOf}(\mathcal{K}, c), \mathcal{F} \uplus (\mathtt{r}, k, v), t^{c'}_{cl}\Big)
\end{aligned}$$

Lemma 17 (Strictly monotonic writers). *Each version for a key has a writer with strictly greater clock time than any versions before:*

$$\forall \mathcal{K}, k, i, j, t^n, t^m.\ \mathsf{w}(\mathcal{K}(k, i)) = t^n \wedge \mathsf{w}(\mathcal{K}(k, j)) = t^m \wedge i < j \Rightarrow n < m$$

By Theorem 16, it is sufficient to only consider normal clock-SI trace. Since transactions do not interleave in a normal clock-SI trace, all transactional execution can be replaced by Fig. 2.

Theorem 17 (Simulation). *Given a clock-SI normal trace τ, a transaction t^n_{cl} from the trace, and the following transactional internal steps*

$$\mathcal{K}_0, c_0, \mathcal{C}_0, s_0, \big[\mathtt{T}\big] \xrightarrow{cl, c, \emptyset, \bot} \cdots \xrightarrow{cl, c, \mathcal{F}, n} \mathcal{K}_i, c_i, \mathcal{C}_i, s_i, \big[\mathtt{skip}\big]$$

for some i, there exists a trace

$$(s_0, \mathsf{snapshot}(\mathcal{K}_0, \mathsf{viewOf}(\mathcal{K}_0, c)), \emptyset), \mathtt{T} \to^* (s_i, \sigma_i, \mathcal{F}_i), \mathtt{skip}$$

that produces the same final fingerprint in the end.

Proof. Given the internal steps of a transaction

$$\mathcal{K}_0, c_0, \mathcal{C}_0, s_0, \left[\mathtt{T}_0\right]_c^{\mathcal{F}_0} \xrightarrow{-} \cdots \xrightarrow{-} \mathcal{K}_i, c_i, \mathcal{C}_i, s_i, \left[\mathtt{T}_i\right]_c^{\mathcal{F}_i}$$

We construct the following trace,

$$(s_0, \mathsf{snapshot}(\mathcal{K}_0, \mathsf{viewOf}(\mathcal{K}_0, c)), \mathcal{F}_0), \mathtt{T}_0 \rightarrow^* (s_i, \sigma_i, \mathcal{F}_i), \mathtt{T}_i$$

Let consider how many transactional internal steps.

– Base case: i = 0. In this case,

$$\mathcal{K}_0, c_0, \mathcal{C}_0, s_0, \left[\mathtt{T}\right]_c^{\mathcal{F}_0}$$

It is easy to construct the following

$$(s_0, \mathsf{snapshot}(\mathcal{K}, \mathsf{viewOf}(\mathcal{K}, c)), \mathcal{F}_0), \mathtt{T}_0$$

– Inductive case: i + 1. Suppose a trace with i steps,

$$\mathcal{K}_0, c_0, \mathcal{C}_0, s_0, \left[\mathtt{T}_0\right]_c^{\mathcal{F}_0} \xrightarrow{-} \cdots \xrightarrow{-} \mathcal{K}_i, c_i, \mathcal{C}_i, s_i, \left[\mathtt{T}_i\right]_c^{\mathcal{F}_i}$$

and a trace

$$(s_0, \mathsf{snapshot}(\mathcal{K}_0, \mathsf{viewOf}(\mathcal{K}_0, c)), \mathcal{F}_0), \mathtt{T}_0 \rightarrow^* (s_i, \sigma_i, \mathcal{F}_i), \mathtt{T}_i$$

Now let consider the next step.

- ReadTrans. In this case

$$\mathcal{K}_i, c_i, \mathcal{C}_i, s_i, \left[\mathtt{T}_i\right]_c^{\mathcal{F}_i} \xrightarrow{-} \mathcal{K}_{i+1}, c_{i+1}, \mathcal{C}_{i+1}, s_{i+1}, \left[\mathtt{T}_{i+1}\right]_c^{\mathcal{F}_{i+1}}$$

such that

$$\mathcal{F}_{i+1} = \mathcal{F}_i \lll (\mathtt{r}, k, v) = \mathcal{F}_i \wedge (\mathtt{w}, k, v) \in \mathcal{F}_i$$

for some key k and value v, and

$$\mathtt{T}_i \equiv \mathtt{x} := [\mathtt{E}]\,; \mathtt{T} \wedge [\![\mathtt{E}]\!]_{s_i} = k \wedge s_{i+1} = s_i[\mapsto v] \wedge \mathtt{T}_{i+1} \equiv \mathtt{skip}; \mathtt{T}$$

for some variable $\mathtt{x}$, expression $\mathtt{E}$ and continuation $\mathtt{T}$. Since $(\mathtt{w}, k, v) \in \mathcal{F}_i$, it means $ß_i(k) = v$ for the local snapshot. By the TPrimitive, we have

$$(s_i, \sigma_i, \mathcal{F}_i), \mathtt{T}_i \rightarrow \mathtt{x} := [\mathtt{E}]\,; \mathtt{T} \rightarrow (s_{i+1}, \sigma_i, \mathcal{F}_{i+1}), \mathtt{T}_{i+1}$$

- ReadRemote. In this case

$$\mathcal{K}_i, c_i, \mathcal{C}_i, s_i, \left[\mathtt{T}_i\right]_c^{\mathcal{F}_i} \xrightarrow{-} \mathcal{K}_{i+1}, c_{i+1}, \mathcal{C}_{i+1}, s_{i+1}, \left[\mathtt{T}_{i+1}\right]_c^{\mathcal{F}_{i+1}}$$

such that

$$\mathcal{F}_{i+1} = \mathcal{F}_i \lll (\mathtt{r}, k, v) = \mathcal{F}_i \uplus \{(\mathtt{r}, k, v)\} \wedge \forall v.\ (\mathtt{w}, k, v') \notin \mathcal{F}_i$$

for some key k and value v, and

$$\mathtt{T}_i \equiv \mathtt{x} := [\mathtt{E}]\,;\mathtt{T} \wedge [\![\mathtt{E}]\!]_{s_i} = k \wedge s_{i+1} = s_i[\mapsto v] \wedge \mathtt{T}_{i+1} \equiv \mathtt{skip};\mathtt{T}$$

for some variable $\mathtt{x}$, expression $\mathtt{E}$ and continuation $\mathtt{T}$. By the premiss of the ReadRemote, the value read is from the last version before the snapshot time:

$$n = \max\left\{n' \;\middle|\; \exists j.\ t^{n'} = \mathsf{w}(\mathcal{K}(k,j)) \wedge n' < c\right\} \wedge \mathsf{val}(\mathcal{K}_i(k,n)) = v$$

By the definition of $u_0 = \mathsf{viewOf}(\mathcal{K}_0, c)$ and $\mathsf{snapshot}(\mathcal{K}_0, u_0)$ and the fact that there is no write to the key k, it follows $\sigma_i(k) = v$. Thus, by the TPrimitive, we have

$$(s_i, \sigma_i, \mathcal{F}_i), \mathtt{T}_i \to \mathtt{x} := [\mathtt{E}]\,;\mathtt{T} \to (s_{i+1}, \sigma_i, \mathcal{F}_{i+1}), \mathtt{T}_{i+1}$$

- Write. In this case

$$\mathcal{K}_i, c_i, \mathcal{C}_i, s_i, \left[\mathtt{T}_i\right]_c^{\mathcal{F}_i} \xrightarrow{-} \mathcal{K}_{i+1}, c_{i+1}, \mathcal{C}_{i+1}, s_{i+1}, \left[\mathtt{T}_{i+1}\right]_c^{\mathcal{F}_{i+1}}$$

such that

$$\mathcal{F}_{i+1} = \mathcal{F}_i \lll (\mathtt{w}, k, v) = \mathcal{F}_i \setminus \{(\mathtt{w}, k, v') \mid v' \in \textsc{Val}\} \uplus \{(\mathtt{r}, k, v)\}$$

for some key k and value v, and

$$\mathtt{T}_i \equiv [\mathtt{E}_1] := \mathtt{E}_2; \mathtt{T} \wedge [\![\mathtt{E}_1]\!]_{s_i} = k \wedge [\![\mathtt{E}_2]\!]_{s_i} = v \wedge \mathtt{T}_{i+1} \equiv \mathtt{skip};\mathtt{T}$$

for some expressions $\mathtt{E}_1$ and $\mathtt{E}_2$, and continuation $\mathtt{T}$. By the TPrimitive, it is easy to see:

$$(s_i, \sigma_i, \mathcal{F}_i), \mathtt{T}_i \to [\mathtt{E}_1] := \mathtt{E}_2; \mathtt{T} \to (s_{i+1}, \sigma_i, \mathcal{F}_{i+1}), \mathtt{T}_{i+1}$$

By Def. 50, Lemma 15, and Theorem 17, we know for each clock-SI trace, there exists a trace that satisfies $\mathtt{ET}_\perp$. Last, we prove such trace also satisfies $\mathtt{ET}_{\mathtt{SI}}$.

Theorem 18 (Clock-SI satisfying SI). *For any normal trace clock-SI trace τ, and transaction t^n_{cl} such that*

$$\tau = \cdots \xrightarrow{cl,c,\mathcal{F},\perp} \mathcal{K}, \mathcal{C}, \mathcal{C}', \mathcal{E}, \mathtt{P} \xrightarrow{cl,c,\mathcal{F},n} \mathcal{K}', \mathcal{C}'', \mathcal{C}''', \mathcal{E}', \mathtt{P}' \xrightarrow{-} \cdots$$

the transaction satisfies $\mathtt{ET}_{\mathtt{SI}}$, i.e. $\mathtt{ET}_{\mathtt{SI}} \vdash (\mathcal{K}, \mathsf{viewOf}(\mathcal{K}, c)) \triangleright \mathcal{F} : \mathsf{viewOf}(\mathcal{K}, \mathcal{C}''(cl))$

Proof. Recall $\mathtt{ET}_{\mathtt{SI}} = \{(\mathcal{K}, u, \mathcal{F}, u') \mid \dagger\} \cap \mathtt{ET}_{\mathtt{MR}} \cap \mathtt{ET}_{\mathtt{RYW}} \cap \mathtt{ET}_{\mathtt{UA}}$ Note that final view of the client, $\mathcal{C}''(cl) = n + 1$. We prove the four parts separately.

– $\{(\mathcal{K}, \mathsf{viewOf}(\mathcal{K}, c), \mathcal{F}, \mathsf{viewOf}(\mathcal{K}, \mathcal{C}''(cl))) \mid \dagger\}$. Assume a version $i \in \mathsf{viewOf}(\mathcal{K}, c)(k)$ for some key k. Suppose a version $\mathcal{K}(k', j)$ such that

$$\mathsf{w}(\mathcal{K}(k', j)) \xrightarrow{((\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}}); \mathsf{RW}^?_{\mathcal{K}})^+} \mathsf{w}(\mathcal{K}(k, i))$$

Let $t^n = \mathsf{w}(\mathcal{K}(k', j))$ and $t^m = \mathsf{w}(\mathcal{K}(k, i))$. By Lemmas 18 and 19, we know $n < m$ then $j \in \mathsf{viewOf}(\mathcal{K}, c)(k')$.

- $\texttt{ET}_{\texttt{MR}}$. By Commit, we know $c \leq n < \mathcal{C}''(cl)$ then $\mathsf{viewOf}(\mathcal{K}, c) \sqsubseteq \mathsf{viewOf}(\mathcal{K}, \mathcal{C}''(cl))$.
- $\texttt{ET}_{\texttt{MW}}$. By Commit, for any write $(\texttt{w}, k, v) \in \mathcal{F}$, there is a new version written by the client cl in the $\mathcal{K}'$,

$$\mathsf{w}(\mathcal{K}'(k, |\mathcal{K}'(k)| - 1)) = t^n_{cl}$$

  Since $n < \mathcal{C}''(cl)$, it follows $|\mathcal{K}'(k)| - 1 \in \mathsf{viewOf}(\mathcal{K}, \mathcal{C}''(cl))(k)$.
- $\texttt{ET}_{\texttt{UA}}$. By the premiss of Commit, for any write $(\texttt{w}, k, v) \in \mathcal{F}$, any existed versions of the key k must be installed by some transactions before the snapshot time of c,

$$\forall k, i.\ (\texttt{w}, k, \_) \in \mathcal{F} \wedge \mathsf{w}(\mathcal{K}(k, i)) < c$$

  It implies that

$$\forall i.\ i \in \mathrm{dom}(\mathcal{K}(k)) \Rightarrow i \in \mathsf{viewOf}(\mathcal{K}, c)(k)$$

Lemma 18 ($\mathsf{RW}_{\mathcal{K}}$). *Given a normal clock-SI trace τ, and two transactions t^n_{cl} and $t^m_{cl'}$ from the trace*

$$\tau = \cdots \dot{\to} \mathcal{K}, \mathcal{C}, \mathcal{C}', \mathcal{E}, \mathtt{P} \xrightarrow{cl, c, \mathcal{F}, n} \cdots \wedge \tau = \cdot \dot{\to} \mathcal{K}', \mathcal{C}'', \mathcal{C}''', \mathcal{E}', \mathtt{P}' \xrightarrow{cl', c', \mathcal{F}', m} \cdots$$

Suppose the final state of the trace τ is $\mathcal{K}''$. , if $t^n_{cl} \xrightarrow{\mathsf{RW}^?_{\mathcal{K}''}} t^m_{cl'}$ then the snapshot time of t^n_{cl} took snapshot before the commit time of $t^m_{cl'}$, i.e. $c \leq m$.

Proof. By definition of $t^n_{cl} \xrightarrow{\mathsf{RW}^?_{\mathcal{K}''}} t^m_{cl'}$, it follows that

$$t^n_{cl} \in \mathsf{rs}(\mathcal{K}''(k, i)) \wedge t^m_{cl'} = \mathsf{w}(\mathcal{K}''(k, j)) \wedge i < j$$

for some key k and indexes i, j. There are two cases depending on the commit order.

- If t^n_{cl} commits after $t^m_{cl'}$, we have,

$$\tau = \cdot \dot{\to} \mathcal{K}', \mathcal{C}'', \mathcal{C}''', \mathcal{E}', \mathtt{P}' \xrightarrow{cl', c', \mathcal{F}', m} \cdots \dot{\to} \mathcal{K}, \mathcal{C}, \mathcal{C}', \mathcal{E}, \mathtt{P} \xrightarrow{cl, c, \mathcal{F}, n} \cdots$$

  We prove by contradiction. Assume $c > m$. Since it is a normal trace τ (Theorem 16), it follows $n > m$. Note that both transactions access the key k, and then by Lemmas 21 and 22, we have $n > c > m$. Given $c > m$, by ReadRemote the transaction t^n_{cl} should at least read the version written by $t^m_{cl'}$ for the key k. That is,

$$t^n_{cl} \in \mathsf{rs}(\mathcal{K}''(k, i)) \wedge t^m_{cl'} = \mathsf{w}(\mathcal{K}''(k, j)) \wedge i > j$$

  which contradict $t^n_{cl} \xrightarrow{\mathsf{RW}^?_{\mathcal{K}''}} t^m_{cl'}$.

– If t^n_{cl} commits before $t^m_{cl'}$,

$$\tau = \cdots \xrightarrow{-} \mathcal{K}, \mathcal{C}, \mathcal{C}', \mathcal{E}, \mathtt{P} \xrightarrow{cl,c,\mathcal{F},n} \cdots \xrightarrow{-} \mathcal{K}', \mathcal{C}'', \mathcal{C}''', \mathcal{E}', \mathtt{P}' \xrightarrow{cl',c',\mathcal{F}',m} \cdots$$

It is trivial that $c \leq m$ by Lemmas 21 and 22.

Lemma 19 ($\mathsf{WR}_\mathcal{K}$, $\mathsf{WW}_\mathcal{K}$ and $\mathsf{SO}_\mathcal{K}$). *Given a normal clock-SI trace τ, and two transactions t^n_{cl} and $t^m_{cl'}$ from the trace*

$$\tau = \cdots \xrightarrow{-} \mathcal{K}, \mathcal{C}, \mathcal{C}', \mathcal{E}, \mathtt{P} \xrightarrow{cl,c,\mathcal{F},n} \cdots \wedge \tau = \cdots \xrightarrow{-} \mathcal{K}', \mathcal{C}'', \mathcal{C}''', \mathcal{E}', \mathtt{P}' \xrightarrow{cl',c',\mathcal{F}',m} \cdots$$

Suppose the final state of the trace τ is $\mathcal{K}''$. , if $t^n_{cl} \xrightarrow{\mathsf{WR}^?_{\mathcal{K}''}} t^m_{cl'}$ then the transaction t^n_{cl} commit before the commit time of $t^m_{cl'}$, i.e. *$n < m$. Similarly, $n < m$ for the relations $\mathsf{WW}_\mathcal{K}$ and $\mathsf{SO}_\mathcal{K}$.*

Proof. – $\mathsf{WR}_{\mathcal{K}''}$. Since $t^n_{cl} \xrightarrow{\mathsf{WR}^?_{\mathcal{K}''}} t^m_{cl'}$, it is only possible that the later commits after the former,

$$\tau = \cdots \xrightarrow{-} \mathcal{K}, \mathcal{C}, \mathcal{C}', \mathcal{E}, \mathtt{P} \xrightarrow{cl,c,\mathcal{F},n} \cdots \wedge \xrightarrow{-} \mathcal{K}', \mathcal{C}'', \mathcal{C}''', \mathcal{E}', \mathtt{P}' \xrightarrow{cl',c',\mathcal{F}',m} \cdots$$

By Lemma 20,we know $n < m$.

– $\mathsf{WW}_{\mathcal{K}''}$. By the definition of $\mathsf{WW}_{\mathcal{K}''}$ and Lemma 17,we know $n < m$.

– $\mathsf{SO}_{\mathcal{K}''}$. By the definition of $\mathsf{SO}_{\mathcal{K}''}$ and Lemma 13,we know $n < m$.

Lemma 20 (Reader greater than writer). *Assume a trace τ and two transactions t^n_{cl} and $t^m_{cl'}$,*

$$\tau = \cdots \xrightarrow{-} \mathcal{K}, \mathcal{C}, \mathcal{C}', \mathcal{E}, \mathtt{P} \xrightarrow{cl,c,\mathcal{F},n} \cdots \wedge \xrightarrow{-} \mathcal{K}', \mathcal{C}'', \mathcal{C}''', \mathcal{E}', \mathtt{P}' \xrightarrow{cl',c',\mathcal{F}',m} \cdots$$

Assume the final state of key-value store of the trace is $\mathcal{K}''$. If $t^m_{cl'}$ reads a version written by t^n_{cl}

$$\mathsf{w}(\mathcal{K}''(k,i)) = t^n \wedge t^m \in \mathsf{rs}(\mathcal{K}''(k,j))$$

Then, the snapshot times of readers of a version is greater then the commit time of the writer $n < c'$

Proof. Trivially, $\mathsf{w}(\mathcal{K}'(k,i)) = t^n$. By the ReadRemote, it follows

$$n = \max\left\{ n' \;\middle|\; \exists j.\ t^{n'} = \mathsf{w}(\mathcal{K}(k,j)) \wedge n' < c' \right\}$$

which implies $n < c'$.

Lemma 21 (Commit time after snapshot time). *The commit time of a transaction is after the snapshot time. Suppose the following step,*

$$\mathcal{K}, \mathcal{C}, \mathcal{C}', \mathcal{E}, \mathtt{P} \xrightarrow{cl,c,\mathcal{F},n} \mathcal{K}', \mathcal{C}'', \mathcal{C}''', \mathcal{E}', \mathtt{P}'$$

then $c < n$.

Proof. It is easy to see by ClientStep and then Commit that

$$n > n - 1 \max\left(\{c' \mid \exists k.\ (\_, k, \_) \in \mathcal{F} \wedge c' = \mathcal{C}'(\mathsf{shardOf}(k))\} \cup \{c\}\right)$$

so $c < n$.

Lemma 22 (Monotonic shard clock time). *The clock time associated with a shard monotonically increases, Suppose the following step,*

$$\mathcal{K}, \mathcal{C}, \mathcal{C}', \mathcal{E}, \mathtt{P} \xrightarrow{cl, c, \mathcal{F}, n} \mathcal{K}', \mathcal{C}'', \mathcal{C}''', \mathcal{E}', \mathtt{P}'$$

then

$$\forall r \in \mathrm{dom}(\mathcal{C}').\ \mathcal{C}'(r) \leq \mathcal{C}'''(r)$$

Proof. We perform case analysis on rules.

- TimeTick By the rule there is one shard r' ticks time $\mathcal{C}'''(r') = \mathcal{C}'(r) + 1 > \mathcal{C}'(r)$.
- ClientStep. There are further five cases, yet only StartTrans and Commit change the shard's clock times.
  - StartTrans By the rule a new transaction starts in a shard r' and triggering the shard r' ticks time $\mathcal{C}'''(r') = \mathcal{C}'(r) + 1 > \mathcal{C}'(r)$.
  - Commit By the rule the transaction commits their fingerprint $\mathcal{F}$ to those shards r' it read or write, and triggering the shard r' ticks time $\mathcal{C}'''(r') = \mathcal{C}'(r) + 1 > \mathcal{C}'(r)$.