Data Consistency in Transactional Storage Systems: A Centralised Semantics

Shale Xiong[1]
Department of Computing, Imperial College London, UK
shale.xiong14@ic.ac.uk

Andrea Cerone[2]
Department of Computing, Imperial College London, UK
andrea.cerone@ic.ac.uk

Azalea Raad
MPI-SWS, Germany
azalea@mpi-sws.org

Philippa Gardner
Department of Computing, Imperial College London, UK
p.gardner@ic.ac.uk

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. Using our abstract states, we present operational definitions of well-known consistency models in the literature, and prove them to be equivalent to their existing declarative definitions using abstract executions. We explore two applications of our operational framework: (1) verifying that the COPS replicated database and the Clock-SI partitioned database satisfy their consistency models using trace refinement, and (2) proving invariant properties of client programs.

2012 ACM Subject Classification Theory of computation → Operational semantics

Keywords and phrases Operational Semantics, Consistency Models, Transactions, Distributed Key-value Stores

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.21

Funding *Shale Xiong*: The Department of Computing, Imperial College London, and EPSRC Fellowship VeTSpec: Verified Trustworthy Software Specification (EP/R034567/1)
Andrea Cerone: EPSRC Programme Grant REMS: Rigorous Engineering for Mainstream Systems (EP/K008528/1), and EPSRC Fellowship VeTSpec: Verified Trustworthy Software Specification (EP/R034567/1)
Azalea Raad: ERC Horizon 2020 Consolidator Grant 'RustBelt' (grant agreement no. 683289)
Philippa Gardner: EPSRC Programme Grant REMS: Rigorous Engineering for Mainstream Systems (EP/K008528/1), and EPSRC Fellowship VeTSpec: Verified Trustworthy Software Specification (EP/R034567/1)

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 known as *consistency models*. Many consistency models were originally

[1] Shale Xiong has moved to Arm Research, `shale.xiong@arm.com`.
[2] Andrea Cerone has moved to Football Radar, `andrea.cerone@footballradar.com`.

invented by engineers using (some quite informal) definitions specific to particular real-world reference implementations, e.g. [3, 4, 6, 8, 21, 33, 38, 42]. More recently, general definitions of consistency model have been defined independently of particular implementations, either declaratively using execution graphs [1, 9] or operationally using abstract states or execution graphs [16, 27, 35]. Our challenge is to define a general semantics for weak consistency models with which we can both verify reference implementations *and* analyse the behaviour of client programs with respect to a particular consistency model.

The declarative approach for defining consistency models using execution graphs has been substantially studied [1, 9, 11, 12, 14]. In such graphs, nodes describe the read-write sets of atomic transactions and edges describe the known dependencies between transactions. They capture different consistency models by: (1) constructing *candidate executions* of the whole program comprising transactions in which reads may contain arbitrary values; and (2) applying the consistency-model *axioms* to rule out candidate executions deemed invalid by the axioms. Such axioms may state, for example, that every read is validated by a write that has written the read value. The most well-known execution graphs are dependency graphs [1] and abstract executions [9, 11]. Dependency graphs tend to be used to analyse client programs, e.g. Fekete et al. [23] derived a static analysis checker for a particular weak consistency model called snapshot isolation; Bernardi and Gotsman [7] developed a static analysis checker for several weak consistency models assuming the so-called snapshot property[3]; and Beillahi et al. [5] developed a tool based on Lipton's reduction theory [31] for checking robustness[4] properties against snapshot isolation. Abstract executions, on the other hand, tend to be used to verify implementation protocols, e.g. abstract executions are the standard by which many system engineers demonstrate that their protocols satisfy certain consistency models [3, 33, 42]. Execution graphs provide little information about how the state evolves throughout the execution of a program, and therefore seem unsuitable for invariant-based program analysis of client programs.

The operational approach for defining weak consistency models has been much less studied. Crooks et al. [16] introduced a trace semantics over abstract centralised kv-stores, abstracting the behaviour of the underlying concrete distributed kv-stores, in order to capture the consistency models associated with ANSI/SQL isolation levels. They describe the equivalence of several implementation-specific definitions of consistency model in the literature, but their reliance on the total transaction order suggests that it will be difficult to adapt their work to reason about client programs. Kaki et al. [27] provide an operational semantics over an abstract centralised store, again focusing on ANSI/SQL isolation levels. They develop a program logic and prototype tool for reasoning about client programs, but cannot express fundamental weak consistency models. Nagar and Jagannathan [35] introduce an operational semantics based on abstract-execution graphs, focussing on consistency models for distributed transactions. They provide robustness results for client programs using model checking, but their analysis is indirect in that they move back and forth between abstract executions and dependency graphs. All these approaches have their merits. However, none provide a direct state-based operational semantics for distributed atomic transactions with which to verify distributed implementations and analyse client programs using the usual weak consistency models; see Section 1.1 for further details on this related work.

---

[3] The *snapshot property*, also known as *atomic visibility*, states that transactional reads appear to read from an atomic snapshot of the database and transactional writes appear to commit atomically, i.e. intermediate transactional states are not observable by clients, even if the underlying distributed protocol has a more fine-grained behaviour.

[4] A particular program (or set of programs) behaves as if the consistency model is serialisability

We introduce an interleaving operational semantics for describing the client-observable behaviour of atomic transactions updating distributed key-value stores (Section 3). Our semantics is based on a notion of abstract states comprising a *centralised key-value store* (kv-store) with multi-versioning and a *client view*. Kv-stores are *global* in that they record all versions of a key; by contrast, client views are *partial* in that a client may see only a subset of the versions. Our client views are partly inspired by the views in the 'promising' C11 semantics [28]. An execution step depends simply on the abstract state, the read-write set of the atomic transaction, and an *execution test*, determining if a client with a given view can 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 models based on abstract executions (reported here and proven in [46]) and thus those based on dependency graphs [14]. Our execution tests are analogous to the commit tests in [16], except that [16] requires analysing the whole trace rather than just the current abstract state.

As in [16, 27, 35], we assume that transactions satisfy the *last-write-wins* resolution policy, a policy widely used in many real-world distributed kv-stores. This means that when a transaction observes several updates to a key, the atomic snapshot contains the value written by the last update. We also assume that our transactions satisfy the *snapshot property*. This is a common assumption in distributed transactional databases, e.g. in online shopping applications, a client only sees one snapshot of the database and only has knowledge that their transaction has successfully committed. The work in [35] also assumes the snapshot property, whereas [16] and [27] do not as their focus is on ANSI/SQL isolation levels [6]. Our execution tests uniformly capture many well-known consistency models (Section 4) including *causal consistency* (`CC`) [9, 33, 40], *parallel snapshot isolation* (`PSI`) [3, 42], *snapshot isolation* (`SI`) [6] and *serialisability* (`SER`) [37]. The work in [35] is as expressive as our work here; by contrast, [16] is more expressive, capturing e.g. the *read committed* consistency model [6], while [27] is less expressive, capturing `SI` but not `PSI`.

Using our operational semantics, we verify that database protocols satisfy their expected consistency models and prove invariant properties of client programs under such consistency models (Section 5). Specifically, we prove the correctness of two database protocols using our general definitions: the COPS protocol for fully replicated kv-stores [33] which satisfies `CC` (reported in Section 5.1 and proved in [46]), and the Clock-SI protocol for partitioned kv-stores [21] which satisfies `SI` (given in [46]). These results had been previously shown for specific consistency definitions devised for the specific reference implementations under consideration. We also prove invariant properties of library clients (Section 5.2): the robustness of the single-counter library against `PSI`, the robustness of the multi-counter library and the banking library [2] against `SI`, and the mutual exclusion of a lock library against `PSI`. We believe our 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 [7] show that multiple counters *are* robust against `PSI` using static-analysis techniques which are known not to be applicable to sessions. These results indicate that our operational semantics provides an interesting abstract interface between distributed databases and clients. This was an important goal for us, resonating with recent work that does just this for standard shared-memory concurrency [17, 19, 25, 36].

1.1 Related Work

Operational semantics for defining weak consistency models for distributed atomic transactions have hardly been studied. To our knowledge, the key papers are [16, 35, 27]. We also mention the log-based semantics of Koskinen and Parkinson [29], which only focuses on serialisability but has some resonance with our work.

Crooks et al. [16] proposed a state-based trace semantics for describing weak consistency models that employs concepts similar to our client views and execution tests, called read states and commit tests respectively. In their semantics, a one-step trace reduction is determined by the entire previous history of the trace. By contrast, our reduction step only depends on the current kv-store and client view. They capture more consistency models than us, e.g. *read committed*, because they do not assume the snapshot property due to their focus on ANSI/SQL isolation levels. They use their semantics to demonstrate that several definitions of snapshot isolation given in the literature [6, 18, 22] in fact collapse into one. They do not verify protocol implementations and do not prove invariant properties of client programs. We believe [16] can be used to verify implementations. We believe it might be difficult to use [16] to prove invariant properties of client programs since their commit tests use total traces. In contrast, our execution tests use partial client views.

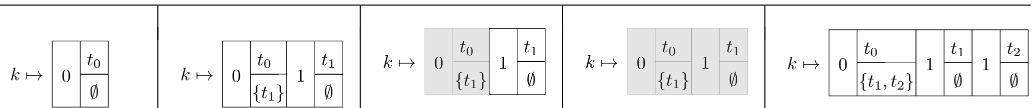
Nagar and Jagannathan [35] proposed a fine-grained interleaving operational semantics on abstract executions, and provide robustness results for client programs using a prototype model-checking tool. They do this by converting abstract executions to dependency graphs and checking the violation of robustness on the dependency graphs. We have two concerns with this approach. First, despite assuming atomic visibility of transactions, they present a fine-grained semantics at the level of the individual transactional operations rather than whole transactions, in order to capture *eventual consistency* [9]. In contrast, our semantics is coarse-grained in that the interleaving is at the level of whole transactions, and we instead capture *read atomic* [4], a variant of *eventual consistency* [9] for atomic transactions. Second, all the literature that performs client analysis on abstract executions [7, 12, 13, 14, 35], including the approach of Nagar and Jagannathan, achieves this indirectly by over-approximating the consistency-model specifications using dependency graphs. It is unknown how to do this precisely [14]. In contrast, we prove robustness results directly by analysing the structure of kv-stores, without over-approximation. We also give precise reasoning about the mutual exclusion of locks, which we believe will be difficult to prove using abstract executions.

Kaki et al. [27] proposed an operational semantics for SQL transactions over an abstract, centralised, single-version store, with consistency models given by the standard ANSI/SQL isolation levels [6]. 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 `SI`, but they do not capture the weaker consistency models such as `PSI` which is an important consistency model for distributed databases. Kaki et al. have explored these weaker consistency models in follow-on work [26], but they focus on an axiomatic semantics for abstract executions over CRDTs not an operational semantics over kv-stores.

Finally, Koskinen and Parkinson [29] proposed a log-based semantics for verifying implementations that satisfy serialisability, based not only on kv-stores but also on other ADTs. Their work comprises a centralised global log and partial client-local logs, similar to our kv-stores and views. Their model focuses on serialisability. There is no evidence that it can be easily extended to tackle weaker consistency models.

2 Overview

We introduce our centralised operational semantics for describing the client-observable behaviours of atomic transactions updating distributed kv-stores. We show that our interleaving semantics provides an abstract interface for both verifying distributed protocols and proving invariant properties of client programs.



(a) Initial kv-store **(b)** the kv-store after t_1 **(c)** A view of cl_2 with the initialisation version **(d)** A view of cl_2 with both versions **(e)** *lost update*: given the view in Figure 1c, the kv-store after t_2

Figure 1 Lost update anomaly: single counter.

Example We use a simple transactional library, `Counter`(k), to introduce our operational semantics. Clients of this library can manipulate the value of counter k via two transactional operations: `Inc (k)` $\triangleq$ [`x` := $[k]$; $[k]$:= `x`+1] and `Read (k)` $\triangleq$ [`x` := $[k]$]. The `x` := $[k]$ reads the value of k in local variable `x`; and $[k]$:= `x`+1 writes `x`+1 to k. The code of each operation is wrapped in square brackets, denoting a transaction that executes *atomically*.

Consider a replicated database where a client only interacts with one replica. For such a database, the behaviour of the atomic transactions is subtle, depending heavily on the particular consistency model under consideration. Consider the client program $\mathtt{P_{LU}}$ below:

$$\mathtt{P_{LU}} \triangleq \quad cl_1 : \mathtt{Inc\,(k)} \mid\mid cl_2 : \mathtt{Inc\,(k)}$$

where we assume that clients cl_1 and cl_2 work on different replicas and, for simplicity, each replica has a kv-store with just one key k. Initially, key k holds value 0 in all replicas. Intuitively, as transactions are executed atomically, after both calls to `Inc (k)` have terminated, the counter should hold value 2. Indeed, this is the only outcome allowed under the *serialisability* (`SER`) consistency model, where transactions appear to execute in a sequential order, one after another. The implementation of `SER` in distributed kv-stores is known to come at a significant performance cost. Implementers are, therefore, content with *weaker* consistency models [3, 6, 8, 21, 32, 33, 38, 42]. For example, if replicas provide no synchronisation mechanism for transactions, it is possible for both clients to read the same initial value 0 for k at their distinct replicas, update it to 1, and eventually propagate their updates of k to other replicas. Thus, both replicas remain unchanged with value 1 for k. This weak behaviour is known as the *lost update* anomaly, which is allowed under *causal consistency* (`CC`), but not under *parallel snapshot isolation* (`PSI`) and *snapshot isolation* (`SI`).

Centralised Operational Semantics Our operational semantics provides transitions over abstract states, comprising a centralised, multi-versioned *kv-store*, which is *global* in that it records all the versions written by all its clients, and a *client view*, which is *partial* in that it records only those versions in the kv-store observed by a client. Each transition of our operational semantics either updates a client-local variable stack using a primitive command, or updates the kv-store and client view using an atomic transaction. The atomic transactions are subject to an *execution test*, which analyses the state to determine if the associated update is allowed under the given consistency model.

We show how the lost update anomaly in $\mathtt{P_{LU}}$ is modelled in our operational semantics. A centralised kv-store provides an abstraction of the real-world replicated key-value store of our example. It is a function mapping keys to a *version* list, recording all the values written to the key together with information about the transactions that accessed it. The total order of versions on a key k is always known due to the resolution policy of the distributed database, for example last-write-wins. In the $\mathtt{P_{LU}}$ example, our initial centralised kv-store comprises a single key k with one initialisation version $(0, t_0, \emptyset)$. This version represents the initialisations in both replicas where k holds value 0, the version *writer* is the initialising transaction t_0 (this version was written by t_0), and the version *reader set* is empty (no transaction has read this version). Figure 1a depicts this initial centralised 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 $\emptyset$.

Suppose that cl_1 first invokes `Inc (k)` on Figure 1a. It does this by choosing a fresh transaction identifier t_1, then reading the initial version of k with value 0 and writing a new value 1 for k. The resulting kv-store is depicted in Figure 1b, where the initial version of k has been updated to reflect that it has been read by t_1 and a new version with value 1 is installed at the end of the list. Now suppose that client cl_2 invokes `Inc (k)` on Figure 1b. As there are now two versions available for k, we must determine the version from which cl_2 fetches its value. This is where the partial *client view* comes 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 key-value stores. In our example, there are two candidate views for cl_2 when running `Inc (k)` on Figure 1b: one containing only the initial version of k as depicted in Figure 1c, and the other containing both versions of k as depicted in Figure 1d[5]. Given the cl_2 view in Figure 1c, client cl_2 chooses a fresh transaction identifier t_2, reads the initial value 0 and writes a new version with value 1, as depicted in Figure 1e. Such a kv-store does not contain a version with value 2, despite two increments on k, producing the lost update anomaly. Had we used the the cl_2 view in Figure 1d instead, client cl_2 would have read the newest value 1 and written a new version with value 2.

The lost update anomaly is allowed under the `CC` consistency model, and disallowed under `SER`, `SI` and `PSI`. To distinguish these cases, we use an *execution test* which directly restricts the updates that are possible at the point where the transaction commits. A simple way of doing this is to require that a client writing a transaction to k have a view containing *all* versions of k available in the global state. This prevents the situation where the view of cl_2 is that given in Figure 1c. This execution test corresponds to what is known in the literature as *write-conflict freedom* [11], which ensures that at most one concurrent transaction can write to a key at any one time.

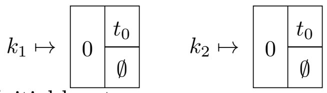
The situation becomes more complicated when the library contains multiple counters where each client can read and increment several counters in one session. For instance, consider the following client program:

$$\begin{aligned}\mathrm{P_{LF}} \triangleq cl_1 : &\left[\mathtt{x} := [k_1]\,; [k_1] := \mathtt{x} + 1\right]; \left[\mathtt{y} := [k_2]\,; [k_2] := \mathtt{y} + 1\right] \\ &\mid\mid cl_2 : \left[\mathtt{x} := [k_1]\,; \mathtt{y} := [k_2]\right] \mid\mid cl_3 : \left[\mathtt{x} := [k_1]\,; \mathtt{y} := [k_2]\right].\end{aligned}$$

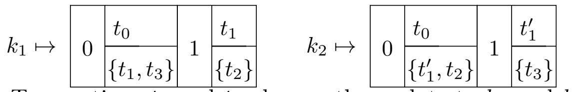
where, for simplicity, the kv-store has just the keys k_1 and k_2 (Figure 2a). Suppose that cl_1 executes both transactions first, writing 1 to k_1 and k_2 using fresh transaction identifiers t_1 and t'_1, respectively. This results in k_1 and k_2 having two versions with values 0 and 1 each, as illustrated in Figure 2b. Client cl_2 next executes its transaction, identified by t_2, using a view that contains both versions of k_1 but only the initial version of k_2. This means that cl_2 reads 1 for k_1 and 0 for k_2, i.e. cl_2 observes the increment of k_1 happening before that of k_2. Symmetrically, cl_3 executes its transaction, identified by t_3, using a view that contains both versions for k_2 but only the initial version of k_1. As such, cl_3 reads 0 for k_1 and 1 for k_2, i.e. cl_3 observes the increment of k_2 happening before that of k_1. This behaviour is known as the *long fork* anomaly (Figure 2b).

The long fork anomaly is disallowed under strong models such as `SER` and `SI`, but is allowed under weaker models such as `PSI` and `CC`. To capture such consistency models and

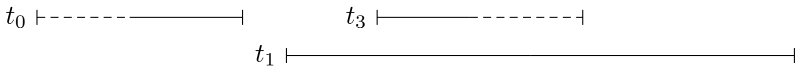
[5] As we explain in Section 3.1, we always require the client view to include the initial version of each key.



(a) Initial kv-store



(b) Transactions t_2 and t_3 observe the update to k_1 and k_2 in different order (*long fork* anomaly)



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

Figure 2 Long fork anomaly: multiple counters

disallow the long fork anomaly of $\mathtt{P_{LF}}$, we must strengthen the execution test associated with the kv-store. For `SER`, we simply strengthen the execution test by ensuring that a client can execute a transaction only if its view contains all versions available in the global state. For `SI`, the execution test is more subtle, requiring that a client view be a set of versions, i.e.*closed* with respect to the commit order of transactions. This means that if a client view includes a version written by a transaction t, then it must include all versions written by transactions that committed before t. Our kv-stores do not contain all the information about the commit order. However, we have enough information to determine the following commit order between transactions: (1) if a transaction, e.g. t_3 in Figure 2, reads a version written by another transaction, e.g. t_0, then it must start after the commit of the transaction that wrote the version, e.g. t_3 must start after the commit of t_0 (Figure 2c); (2) if a transaction writes a newer version of a key, e.g. t_1 for k_1, then it must commit after the transactions that wrote the previous versions of the key,e.g. t_0 (Figure 2c); and (3) if a transaction reads an older version of a key, e.g. t_3 for k_1, it must start before the commit of all transactions that write the newer versions of k, e.g. t_1 (Figure 2c).

In Section 4, we formally define the execution tests associated with several consistency models on kv-stores and client views. In [46], we show the equivalence of our operational definitions of consistency models and the existing declarative definitions based on abstract executions [11], and hence those based on dependency graphs [1].

Verifying Implementation Protocols The first application of our operational semantics is to show that implementation protocols of distributed key-value stores satisfy certain consistency models. We do this by representing the implementation protocol using our centralised operational semantics: our abstract states provide a faithful abstraction of replicated and partitioned databases, and our execution tests provide a faithful abstraction of the synchronisation mechanisms enforced by these databases when committing a transaction. We verify the correctness of our representation using trace refinement. Thus, a distributed protocol satisfies the particular consistency model associated with the particular execution test of our representation. We demonstrate that the COPS protocol [33] for implementing a replicated database satisfies our definition of `CC` (reported in Section 5.1 and proved in [46]), and the Clock-SI protocol [21] for implementing a partitioned database satisfies our definition of `SI` (given in [46]). Since our definitions of consistency model are equivalent to those in the literature [46], we have demonstrated that COPS and Clock-SI satisfy the accepted general definitions of the respective consistency models. This contrasts with the previous results in [33] and [21] which demonstrated that these protocols satisfy specific consistency models defined for those particular implementations.

Proving Invariant Properties of Client Programs The second application of our operational semantics is to prove invariant properties for transactional libraries (Section 5.2). One well-known property is *robustness*. A library is robust against a (weak) consistency model `M` if, for all its client programs `P` and all kv-stores $\mathcal{K}$, if $\mathcal{K}$ is obtained by executing `P` under `M`, then $\mathcal{K}$ can also be obtained under `SER`, i.e. library clients have no observable weak behaviours. We prove the robustness of the single counter library against `PSI`, and the robustness of a multi-counter library and the banking library of [2] against `SI`. We prove robustness against `SI` by proving general invariants that guarantee robustness against a new model we propose, `WSI`, which lies between `PSI` and `SI`. As we discuss in Section 5.2, although existing techniques [35, 12, 7] in the literature can verify such robustness properties, they typically do so by examining *full traces*. By contrast, we establish invariant properties at each execution step of our operational semantics, thus allowing a simpler, more compositional proof.

We also demonstrate the use of our operational semantics to prove library-specific invariant properties. In particular, we show that a lock library is correct against `PSI`, in that it satisfies the *mutual exclusion guarantee*, even though it is not robust against `PSI`. To do this, we encode this guarantee as an invariant of the lock library, establishing the invariant at each transition step of the operational semantics. By contrast, establishing such library-specific properties using the existing techniques is more difficult. This is because existing techniques [35, 12] do not directly record the library *state*; rather, they record full execution traces, making them less amenable for reasoning about such properties.

3 Operational Model

We define an interleaving operational semantics for atomic transactions (Section 3.2) on abstract states comprising global kv-stores and partial client views (Section 3.1). Our semantics is parametrised by an execution test which induces a consistency model (Section 4).

3.1 Abstract States: Key-Value Stores and Client Views

The abstract states of our operational semantics comprise a global, centralised kv-store and a partial client view. 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*[6], $\textsc{ClientID} \ni cl$. The set of *transaction identifiers*, $\textsc{TxID} \ni t$, is defined by $\textsc{TxID} \triangleq \{t_0\} \uplus \{t_{cl}^n \mid cl \in \textsc{ClientID} \land n \geq 0\}$, where t_0 denotes the *initialisation transaction* and t_{cl}^n identifies a transaction committed by client cl with n determining the client session order: $\mathsf{SO} \triangleq \{(t, t') \mid \exists cl, n, m.\, t = t_{cl}^n \land t' = t_{cl}^m \land n < m\}$. Subsets of $\textsc{TxID}$ are ranged over by $T, T', \cdots$. We let $\textsc{TxID}_0 \triangleq \textsc{TxID} \setminus \{t_0\}$.

▶ **Definition 1** (Kv-stores). *Assume a countably infinite set of* keys, $\textsc{Key} \ni k$, *and a countably infinite set of* values, $\textsc{Value} \ni v$, *which includes the keys and an* initialisation value v_0. *The set of* versions, $\textsc{Version} \ni \nu$, *is* $\textsc{Version} \triangleq \textsc{Value} \times \textsc{TxID} \times \mathcal{P}(\textsc{TxID}_0)$. *A* kv-store *is a function* $\mathcal{K} : \textsc{Key} \to \mathsf{List}(\textsc{Version})$, *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 write $\mathsf{val}(\nu)$, $\mathsf{w}(\nu)$ and $\mathsf{rs}(\nu)$ to project the 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 in $\mathcal{K}$; we write $|\mathcal{K}(k)|$ for the length of the version list $\mathcal{K}(k)$, and $\mathcal{K}(k, i)$ for the i^{th} version of k in kv-store $\mathcal{K}$.

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

We assume that the version list of 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:

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

This is a standard assumption for distributed databases, e.g. in [3, 4, 6, 8, 21, 33, 38, 42]. Finally, we assume that the kv-store agrees with the session order of clients, in that 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, i.e. for any k, i, j, t, t':

$$t = \mathsf{w}(\mathcal{K}(k,i)) \wedge t' \in \mathsf{rs}(\mathcal{K}(k,i)) \Rightarrow (t', t) \notin \mathsf{SO}^{?} \qquad \text{(wr-so)}$$

$$t = \mathsf{w}(\mathcal{K}(k,i)) \wedge t' = \mathsf{w}(\mathcal{K}(k,j)) \wedge i < j \Rightarrow (t', t) \notin \mathsf{SO}^{?} \qquad \text{(ww-so)}$$

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.

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 *client view*, or just *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. This client view was partly inspired by the views of the 'promising' C11 operational semantics [28].

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

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

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

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

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 $\mathcal{K} \in KVS$ 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 $k \in \textsc{Key}$.*

Given a configuration $(\mathcal{K}, \mathcal{U})$ and a client cl, if $u = \mathcal{U}(cl)$ is defined then, for each k, the configuration determines the sub-list of versions in $\mathcal{K}$ that cl sees. If $i, j \in u(k)$ 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 $\mathcal{K} \in KVS$ and $u \in \textsc{Views}(\mathcal{K})$, the* view snapshot *of u in $\mathcal{K}$ is a function,* $\mathsf{snapshot}(\mathcal{K}, u) : \textsc{Key} \to \textsc{Value}$*, 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 $u(k)$ under the natural order $<$ on $\mathbb{N}$.

When clear from the context, we simply refer to a view snapshot as a *snapshot*.

3.2 Operational Semantics

Core 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: $[\![v]\!]_s \triangleq v \quad [\![\mathtt{x}]\!]_s \triangleq s(\mathtt{x}) \quad [\![\mathtt{E}_1 + \mathtt{E}_2]\!]_s \triangleq [\![\mathtt{E}_1]\!]_s + [\![\mathtt{E}_2]\!]_s \quad \cdots$. 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* $\mathtt{C}$, defined by:

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

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

Sequential commands ($\mathtt{C}$) comprise $\mathtt{skip}$, primitive commands ($\mathtt{C}_p$), atomic transactions ($\left[\mathtt{T}\right]$), 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 ($\mathtt{T}$) comprises $\mathtt{skip}$, primitive transactional commands ($\mathtt{T}_p$), and the standard compound constructs. Primitive transactional commands comprise primitive commands as well as lookup ($\mathtt{x} := [\mathtt{E}]$) and mutation ($[\mathtt{E}] := \mathtt{E}$) used, respectively, to read and write a single key to a kv-store, and can only be invoked within an atomic transaction.

A *program* $\mathtt{P}$ is a finite partial function from client identifiers to sequential commands. For clarity, we often write $\mathtt{C}_1 \parallel \ldots \parallel \mathtt{C}_n$ for a program with n clients identified by $cl_1 \ldots cl_n$, with each client cl_i executing $\mathtt{C}_i$. Each client cl_i is associated with a client-local *stack*, $s_i \in \textsc{Stack} \triangleq \textsc{Var} \to \textsc{Value}$, 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, e.g. COPS [33], to update the underlying distributed kv-stores while the transaction is in progress. It just means that, given the abstractions captured by our global kv-stores and partial client 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 $\left[\mathtt{T}\right]$, it performs the following steps: (1) it constructs an initial *snapshot* σ of $\mathcal{K}$ using its view $\mathcal{U}(cl)$ as described in Definition 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*, $\sigma \in \textsc{Snapshot} \triangleq \textsc{Key} \to \textsc{Value}$, is a function from keys to values.*

When clear from the context, we simply refer to a transactional snapshot as a *snapshot*.

The rules for transactional commands (Figure 3) are defined using an arbitrary transactional snapshot. The rules for sequential commands and programs (Figure 4) are 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 first values $\mathtt{T}$ reads from σ, and the last values $\mathtt{T}$ writes to σ and intends to commit to $\mathcal{K}$. Execution of a transaction in a given configuration and variable stack may result in more than one fingerprint due to non-determinism (non-deterministic choice).

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

$$\begin{aligned}
\mathcal{F} \lll (\mathtt{R},k,v) &\triangleq \begin{cases}\mathcal{F}\cup\{(\mathtt{R},k,v)\} & \text{if } \forall l,v'.\,(l,k,v')\notin\mathcal{F}\\ \mathcal{F} & \text{otherwise}\end{cases}\\
\mathcal{F} \lll (\mathtt{W},k,v) &\triangleq (\mathcal{F}\setminus\{(\mathtt{W},k,v') \mid v'\in\textsc{Value}\})\cup\{(\mathtt{W},k,v)\}\\
\mathcal{F} \lll \epsilon &\triangleq \mathcal{F}
\end{aligned}$$

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

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

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

$$\mathsf{op}\,(s,\sigma,\mathtt{x} := [\mathtt{E}]) \triangleq (\mathtt{R},[\![\mathtt{E}]\!]_s,\sigma([\![\mathtt{E}]\!]_s)) \qquad \mathsf{op}\,(s,\sigma,[\mathtt{E_1}] := \mathtt{E_2}) \triangleq (\mathtt{W},[\![\mathtt{E_1}]\!]_s,[\![\mathtt{E_2}]\!]_s)$$

Figure 3 The semantics of transactional commands

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

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

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

Figure 4 The semantics of sequential commands and programs

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

A fingerprint 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 rule for primitive transactional commands, TPrimitive, is given in Figure 3. The rules for the compound constructs are straightforward and given in [46]. 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 $\mathtt{T}_p$, where the empty operation ϵ is used for those primitive commands that do not contribute to the fingerprint.

The conclusion of TPrimitive uses the *combination operator* $\lll : \mathcal{P}(\textsc{Op})\times(\textsc{Op}\uplus\{\epsilon\})\to\mathcal{P}(\textsc{Op})$, defined in Figure 3, 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 Figure 4. 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 $\mathcal{K}$, client 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 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 the *execution test* $\mathtt{ET}$, which is used to generate the *consistency model* under which a transaction can execute. In Section 4, we give several examples of execution tests for well-known consistency models. In [46], we prove that the consistency models generated by our execution tests are equivalent to their corresponding existing definitions using abstract executions.

The rules for compound constructs are straightforward and given in [46]. The rule for primitive commands, CPrimitive, depends on the transition system $\overset{\mathtt{C}_p}{\rightsquigarrow} \subseteq \textsc{Stack} \times \textsc{Stack}$ which describes how the primitive command $\mathtt{C}_p$ affects the stack. The CAtomicTrans rule describes the execution of an atomic transaction under the execution test $\mathtt{ET}$.

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 Definition 2). Given the new view u'', the transaction obtains a snapshot σ of the kv-store $\mathcal{K}$, and executes $\mathtt{T}$ locally to completion ($\mathtt{skip}$), updating the stack to s', while accumulating the fingerprint $\mathcal{F}$, as described by 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 $\mathtt{canCommit}_{\mathtt{ET}}(\mathcal{K}, u'', \mathcal{F})$ premise ensures that, under the execution test $\mathtt{ET}$, 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*. Observe that the $\mathtt{canCommit}$ check is parametric in the execution test $\mathtt{ET}$. This is because the conditions checked upon committing depend on the consistency model under which the transaction is to commit. In Section 4, we define $\mathtt{canCommit}$ for several execution tests associated with well-known consistency models.

Client cl is now ready to commit the transaction resulting in the kv-store $\mathcal{K}'$ with the client view u'' *shifting* to a new view u' and proceeds as follows: (1) it picks a fresh transaction identifier $t \in \mathsf{NextTxID}(cl, \mathcal{K})$; (2) computes the new kv-store $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, u'', \mathcal{F}, t;$ and (3) checks if the *view shift* is permitted under $\mathtt{ET}$ using $\mathtt{vShift}_{\mathtt{ET}}(\mathcal{K}, u'', \mathcal{K}', u')$. Note that as with $\mathtt{canCommit}$, the $\mathtt{vShift}$ check is parametric in the execution test $\mathtt{ET}$. This is because the conditions checked for shifting the client view depend on the consistency model. In Section 4 we define $\mathtt{vShift}$ for several execution tests associated with well-known consistency models. The set $\mathsf{NextTxID}(cl, \mathcal{K})$ is given by: $\{t_{cl}^{n} \mid \forall m.\, t_{cl}^{m} \in \mathcal{K} \Rightarrow m < n\}$. The function $\mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t)$ 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 $\mathcal{K}(k)$. The function $\mathsf{UpdateKV}$ is well-formed, because a fingerprint contains at most one write operation and one read operation for a given key (see [46] for the full details).

▶ **Definition 7** (Transactional update). *The function* $\mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t)$ *is defined as:*

$$\mathsf{UpdateKV}(\mathcal{K}, u, \emptyset, t) \triangleq \mathcal{K}$$

$$\begin{aligned}\mathsf{UpdateKV}(\mathcal{K}, u, \{(\mathtt{R}, k, v)\} \uplus \mathcal{F}, t) &\triangleq \textit{let } i = \max_{<}(u(k)) \textit{ and } (v, t', T) = \mathcal{K}(k, i) \textit{ in} \\ &\qquad \mathsf{UpdateKV}\left(\mathcal{K}\left[k \mapsto \mathcal{K}(k)\left[i \mapsto (v, t', T \uplus \{t\})\right]\right], u, \mathcal{F}, t\right)\end{aligned}$$

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

where $\mathcal{V}[i \mapsto \nu] \triangleq \nu_0 :: \cdots :: \nu_{i-1} :: \nu :: \nu_{i+1} :: \cdots :: \nu_n$ *for all version lists* $\mathcal{V} = \nu_0 :: \cdots :: \nu_n$ *and indexes* $i : 0 \leq i \leq n$.

The last rule, PProg (Figure 4), captures the execution of a program step using a *client environment*, $\mathcal{E} \in \textsc{CEnv}$, which is a function from client identifiers to stacks associating each client with its stack. We assume that the domain of a 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 Using Execution Tests on 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, e.g. [3, 6, 8, 21, 32, 33, 38, 42], which capture different trade-offs between performance and application correctness. Example consistency models range from *serialisability*, a strong model which only allows kv-stores obtained from a serial execution of transactions with inevitable performance drawbacks, to *eventual consistency*, a weak model which imposes few conditions on the structure of kv-stores, leading to good performance but anomalous behaviours. We define consistency models for our kv-stores, by introducing the notion of an *execution test*, specifying whether a client is allowed to commit a transaction in a given kv-store. An execution test ET induces a consistency model as the set of kv-stores obtained by having clients non-deterministically commit transactions, so long as the constraints imposed by ET are satisfied. We explore a range of execution tests associated with well-known consistency models in the literature. In [46], we demonstrate that our operational definitions of consistency models over kv-stores using execution tests are equivalent to the established declarative definitions of consistency models over abstract executions [9, 11].

▶ **Definition 8** (Execution tests). *An* execution test, $\mathtt{ET}$*, is a set of tuples,* $\mathtt{ET} \subseteq KVS \times \textsc{Views} \times \textsc{Fp} \times KVS \times \textsc{Views}$*, such that for all* $(\mathcal{K}, u, \mathcal{F}, \mathcal{K}', u') \in \mathtt{ET}$*: (1)* $u \in \textsc{Views}(\mathcal{K})$ *and* $u' \in \textsc{Views}(\mathcal{K}')$*; (2)* $\mathtt{canCommit}_{\mathtt{ET}}(\mathcal{K}, u, \mathcal{F})$*; (3)* $\mathtt{vShift}_{\mathtt{ET}}(\mathcal{K}, u, \mathcal{K}', u')$*; and (4) for all* $k \in \mathcal{K}$ *and* $v \in \textsc{Value}$*, if* $(\mathtt{R}, k, v) \in \mathcal{F}$ *then* $\mathcal{K}(k, \max_{<}(u(k))) = v$*.*

Intuitively, $(\mathcal{K}, u, \mathcal{F}, \mathcal{K}', u') \in \mathtt{ET}$ 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 $\mathcal{K}'$ (given by Definition 7) while shifting its view to u'. Note that the last condition in Definition 8 enforces the last-write-wins policy [45]: a transaction always reads the most recent writes from the initial view u.

▶ **Definition 9** (Consistency models). *The* consistency model *induced by an execution test* ET *is defined as:* $\mathsf{CM}(\mathtt{ET}) \triangleq \left\{\mathcal{K} \mid \exists \mathcal{K}_0, \mathcal{U}_0, \mathcal{E}, \mathtt{P}.\, (\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}), \mathtt{P} \Rightarrow^{*}_{\mathtt{ET}} (\mathcal{K}, \_, \_), \_\right\}$.

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

$$\mathtt{canCommit}_{\mathtt{ET}_\top}(\mathcal{K}, u, \mathcal{F}) \overset{\text{def}}{\Leftrightarrow} \mathsf{true} \qquad \text{and} \qquad \mathtt{vShift}_{\mathtt{ET}_\top}\left(\mathcal{K}, u, \mathcal{K}', u'\right) \overset{\text{def}}{\Leftrightarrow} \mathsf{true}$$

The consistency model induced by $\mathtt{ET}_\top$ corresponds to the *Read Atomic* model [4], a variant of *Eventual Consistency* [9] for atomic transactions.

We present several examples of execution tests which give rise to consistency models on kv-stores. Recall that the snapshot property and the last-write-wins policy are hard-wired in our framework. As such, we can only define consistency models that satisfy these two constraints. Although this prohibits interesting consistency models such as *Read Committed*, we can express a large number 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 the reflexive, transitive and reflexive-transitive closures of r, respectively; r^{-1} for the inverse of r; $a_1 \xrightarrow{\mathsf{r}} a_2$ for $(a_1, a_2) \in \mathsf{r}$; and $\mathsf{r}; \mathsf{r}'$ for $\{(a_1, a_2) \mid \exists a.\, (a_1, a) \in \mathsf{r} \wedge (a, a_2) \in \mathsf{r}'\}$.

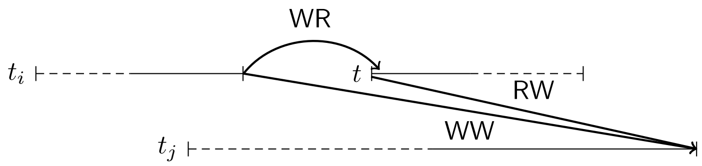


Figure 5 An example of dependencies between transactions with respect to the time line of the starts and commits of these transactions (dashed line being able to stretched)

Recall that an execution test ET is a tuple $(\mathcal{K}, u, \mathcal{F}, \mathcal{K}', u')$ such that $\texttt{canCommit}_{\texttt{ET}}(\mathcal{K}, u, \mathcal{F})$ and $\texttt{vShift}_{\texttt{ET}}(\mathcal{K}, u, \mathcal{K}', u')$ hold (Definition 8). We proceed with several auxiliary definitions that allow us to define `canCommit` and `vShift` for several consistency models.

Prefix Closure The *set of visible transactions* of a kv-store $\mathcal{K}$ and a view u is: $\mathsf{visTx}(\mathcal{K}, u) \triangleq \{\mathsf{w}(\mathcal{K}(k, i)) \mid i \in u(k)\}$. Given a relation on transactions, $R \subseteq \text{TxID} \times \text{TxID}$, a view u is *closed* with respect to a kv-store $\mathcal{K}$ and R, written $\texttt{closed}(\mathcal{K}, u, R)$, if and only if:

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

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

Dependency Relations We next define transactional dependency relations for kv-stores. Figure 7a illustrates an example kv-store and its transactional dependency relations. Given a kv-store $\mathcal{K}$, a key k and indexes i, j such that $0 \leq i < j < |\mathcal{K}(k)|$, if there exists t_i, T_i, t such that $\mathcal{K}(k, i) = (\_, t_i, T_i)$, $\mathcal{K}(k, j) = (\_, t_j, \_)$ and $t \in T_i$, then for every key k:

(1) there is a *Write-Read* dependency from t_i to t, written $(t_i, t) \in \mathsf{WR}_{\mathcal{K}}(k)$, which intuitively means that t_i commits before t starts, as depicted in Figure 5;
(2) there is a *Write-Write* dependency from t_i to t_j, written $(t_i, t_j) \in \mathsf{WW}_{\mathcal{K}}(k)$, which intuitively means that t_i commits before t_j commits, as depicted in Figure 5; and
(3) if $t \neq t_j$, then there is a *Read-Write* anti-dependency from t to t_j, written $(t, t_j) \in \mathsf{RW}_{\mathcal{K}}(k)$, which intuitively means that t starts before t_j commits, as depicted in Figure 5.

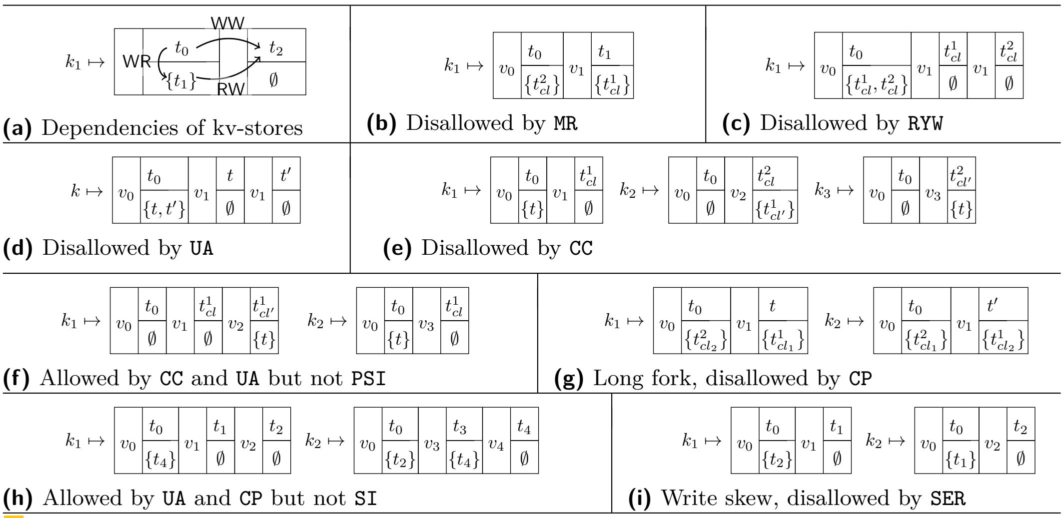
In centralised databases, where there is a global notion of time, these dependency relations can be determined by the start and commit time of transaction as in Figure 5. However, in general, there is no global notion of time in distributed databases. In such settings, the write-read dependency WR is induced when a transaction reads from another transaction; the write-write dependency WW is given by the *last-write-wins* resolution policy, ordering the transactions that write to the same key; and the read-write anti-dependency RW is derived from WR and WW: if $(t, t') \in \mathsf{WR}$ and $(t, t'') \in \mathsf{WW}$, then $(t', t'') \in \mathsf{RW}$. We adopt the same names as the dependency relations of dependency graphs [1] to underline the similarity. However, our relations here do *not* depend on those relations in dependency graphs.

We give several definitions of execution tests using `vShift` and `canCommit` in Figure 6.

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 Figure 7b, since client cl first reads the latest version of k in t_{cl}^1, and then reads the older, initial version of k in t_{cl}^2. As such, the $\texttt{vShift}_{\texttt{MR}}$ predicate in Figure 6 ensures that clients can only extend their views. When this is the case, clients can *always* commit their transactions, and thus $\texttt{canCommit}_{\texttt{MR}}$ is simply `true`.

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

Figure 6 Execution tests of consistency models defined by `canCommit` and `vShift` predicates, where SO is as given in Section 3.1.



(a) Dependencies of kv-stores **(b)** Disallowed by `MR` **(c)** Disallowed by `RYW` **(d)** Disallowed by `UA` **(e)** Disallowed by `CC` **(f)** Allowed by `CC` and `UA` but not `PSI` **(g)** Long fork, disallowed by `CP` **(h)** Allowed by `UA` and `CP` but not `SI` **(i)** Write skew, disallowed by `SER`

Figure 7 Behaviours disallowed under different consistency models. Figure 7a shows the dependencies of transactions in kv-stores (values omitted).

Read Your Writes (`RYW`) This consistency model states that a client must always see all the versions written by the client itself. The $\texttt{vShift}_{\texttt{RYW}}$ predicate thus states that after executing a transaction, a client contains all the versions it wrote in its view. This ensures that such versions will be included in the view of the client when committing future transactions. Note that under `RYW` the kv-store in Figure 7c 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_{cl}^1, it reads the initial value v_0 and then writes a new version with value v_1. For its second transaction t_{cl}^2, it reads the initial value v_0 again and writes a new version with value v_1. The $\texttt{vShift}_{\texttt{RYW}}$ predicate rules out this example by requiring the client view after committing t_{cl}^1 to include the version it wrote. When this is the case, clients can always commit their transactions, and thus $\texttt{canCommit}_{\texttt{RYW}}$ is simply `true`.

The `MR` and `RYW` models, together with the *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 given in [46].

We now give the definitions of well-known consistency models in distributed databases, including `CC` [9, 33, 40], `PSI` [3, 42], `SI` [6] and `SER` [37]. The `vShift` relation for these consistency models, given in Figure 6, is simply $\texttt{vShift}_{\texttt{MR}\cap\texttt{RYW}}(\mathcal{K}, u.\mathcal{K}', u') = \texttt{vShift}_{\texttt{MR}}(\mathcal{K}, u.\mathcal{K}', u') \cap \texttt{vShift}_{\texttt{RYW}}(\mathcal{K}, u.\mathcal{K}', u')$. The $\texttt{canCommit}_{\texttt{ET}}\,(\mathcal{K}, u, \mathcal{F})$ relation is defined by $\texttt{canCommit}_{\texttt{ET}}\,(\mathcal{K}, u, \mathcal{F}) \triangleq \texttt{closed}(\mathcal{K}, u, R_{\texttt{ET}})$ where $R_{\texttt{ET}}$ is given for each exection test in Figure 6 as a combination of SO and the dependency relations. We use two less-known consistency models, *update atomic* (UA) and *consistent prefix* (CP). In [7, 10, 11], the definition of SI on abstract executions can be separated into the conjunction of UA and CP. Similarly, the definition of PSI on abstract executions can be separated into the conjunction of UA and CC [11]. Interestingly, this is not quite the case for the consistency definitions presented here.

Causal Consistency (CC) This model states that, if a client view includes a version ν written by t prior to committing a transaction, then it must also include the versions which t observes. Clearly, t observes all versions that t reads. Moreover, t observes all previous transactions from the same client. This is captured by $\texttt{canCommit}_{\texttt{CC}}$ in Figure 6, defined as $\texttt{closed}(\mathcal{K}, u, R_{\texttt{CC}})$ with $R_{\texttt{CC}} \triangleq \mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}$. For example, the kv-store of Figure 7e is disallowed by CC: the k_3 version with value v_3 depends on the k_1 version with value v_1. However, 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 in [11] and implemented in [32]. 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 $\texttt{canCommit}_{\texttt{UA}}$ 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 Figure 7d, 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.

Parallel Snapshot Isolation (PSI) This consistency model states that: (1) if a client view includes a version ν written by t prior to committing a transaction, then it must also include the versions that t observes; and (2) there are no write-conflicts.

On abstract executions, where there is a total order over transactions, PSI can be formally defined as the composition of CC and UA [11]. By contrast, it is not possible to define $\texttt{canCommit}_{\texttt{PSI}}$ as the conjunction of the $\texttt{canCommit}_{\texttt{CC}}$ and $\texttt{canCommit}_{\texttt{UA}}$ relations. This is for two reasons. First, the conjunction would only mandate that u be closed with respect to $R_{\texttt{CC}}$ and $R_{\texttt{UA}}$ 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 $R_{\texttt{CC}}$ and $R_{\texttt{UA}}$ is smaller than the closure of $R_{\texttt{CC}} \cup R_{\texttt{UA}}$. As such, we define $\texttt{canCommit}_{\texttt{PSI}}$ as closure with respect to $R_{\texttt{PSI}}$ which includes $R_{\texttt{CC}} \cup R_{\texttt{UA}}$. Second, recall that CC requires that if a client view includes a version ν written by t' prior to committing a transaction, then it must also include the versions which t' observes. For example, the view of the client of transaction t in Figure 7f must include versions written by t_0 and $t^1_{cl'}$, satisfying $\texttt{canCommit}_{\texttt{CC}}$. Also, recall that UA requires that if a transaction writes to a key k then it must observe all previous versions of k. For example, the client cl' that writes the third version of k_1 in Figure 7f must observe t^1_{cl}, satisfying $\texttt{canCommit}_{\texttt{UA}}$. However, although the client of transaction t observes $t^1_{cl'}$, it is not able to observe t^1_{cl} using the combination of CC and UA. This is fixed by including the the write-write dependency relation $\mathsf{WW}_{\mathcal{K}}$ (e.g. $(t^1_{cl}, t^1_{cl'}) \in \mathsf{WW}_{\mathcal{K}}$) in $R_{\texttt{PSI}}$. Note that Figure 7f shows an example kv-store that satisfies $\texttt{canCommit}_{\texttt{CC}}$ and $\texttt{canCommit}_{\texttt{UA}}$, but not $\texttt{canCommit}_{\texttt{PSI}}$. Under PSI, the view of the client of t should include the versions written by t^1_{cl}, and therefore read v_3 for key k_2.

Consistent Prefix (CP) If the total order in which transactions commit is known, then CP can be described as a strengthening of CC [14]: 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 order of transaction commits, this is sufficient to formalise $\texttt{CP}$.

We can approximate the order in which transactions commit using $\mathsf{WR}_{\mathcal{K}}, \mathsf{WW}_{\mathcal{K}}, \mathsf{RW}_{\mathcal{K}}$ and SO. This approximation is perhaps best understood in terms of an idealised implementation of $\texttt{CP}$ 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*. In 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', i.e. 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 $R_{\texttt{CP}} \triangleq (\mathsf{WR}_{\mathcal{K}}; \mathsf{RW}_{\mathcal{K}}^{?} \cup \mathsf{SO}; \mathsf{RW}_{\mathcal{K}}^{?} \cup \mathsf{WW})$, approximating the order in which transactions commit. As shown in [14], the set $(R_{\texttt{CP}}^{+})^{-1}(t)$ contains all transactions that must be observed by t under $\texttt{CP}$. We thus define $\texttt{canCommit}_{\texttt{CP}}$ by requiring closure with respect to $R_{\texttt{CP}}$.

The $\texttt{CP}$ model disallows the *long fork anomaly* in Figure 7g, where 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 cl_2 sees the k_1 version with value v_0 before committing $t^2_{cl_2}$. However, as $t \xrightarrow{\mathsf{WR}_{\mathcal{K}}} t^1_{cl_1} \xrightarrow{\mathsf{SO}} t^2_{cl_1} \xrightarrow{\mathsf{RW}} t' \xrightarrow{\mathsf{WR}} t^1_{cl_2}$ and $t^2_{cl_2}$ must see the versions written by $t^1_{cl_2}$ before committing, then $t^2_{cl_2}$ must also see the k_1 version with value v_2, leading to a contradiction.

Snapshot Isolation ($\texttt{SI}$) On abstract executions, where there is a total order over transactions, $\texttt{SI}$ can be defined as the composition of $\texttt{CP}$ and $\texttt{UA}$. However, as with $\texttt{PSI}$, we cannot define $\texttt{canCommit}_{\texttt{SI}}$ as the conjunction of their associated $\texttt{canCommit}$ predicates. Rather, we define $\texttt{canCommit}_{\texttt{SI}}$ as closure with respect to $R_{\texttt{SI}}$ which includes $R_{\texttt{CP}} \cup R_{\texttt{UA}}$. Observe that Figure 7h shows an example kv-store that satisfies $\texttt{canCommit}_{\texttt{UA}}$ and $\texttt{canCommit}_{\texttt{CP}}$, but not $\texttt{canCommit}_{\texttt{SI}}$. Additionally, we include $\mathsf{WW}; \mathsf{RW}$ in $R_{\texttt{SI}}$. This is because, when the centralised $\texttt{CP}$ implementation (discussed before) is strengthened with write-conflict freedom, then a write-write dependency between transactions t and t' does not only mandate that t commit before t' commits, but also before t' starts. Consequently, if $(t, t') \in \mathsf{WW}; \mathsf{RW}$, then t must commit before t' does.

(Strict) serialisability ($\texttt{SER}$) Serialisability is the strongest consistency model in settings that abstract from aborted transactions, requiring that transactions execute in a total sequential order. The $\texttt{canCommit}_{\texttt{SER}}$ 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 Figure 7i: if, without loss of generality, t_1 commits before t_2, then the client committing t_2 must see the k_1 version written by t_1, and thus cannot read the outdated value v_0 for k_1.

Weak Snapshot Isolation ($\texttt{WSI}$)**: 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 $\texttt{CP}$ and $\texttt{UA}$, which we refer to as *Weak Snapshot Isolation* ($\texttt{WSI}$). Because $\texttt{WSI}$ is stronger than $\texttt{CP}$ and $\texttt{UA}$ by definition, it forbids all the anomalies forbidden by these consistency models, e.g. the long fork (Figure 7g) and the lost update (Figure 7d). Moreover, $\texttt{WSI}$ is strictly weaker than $\texttt{SI}$. As such, $\texttt{WSI}$ allows all $\texttt{SI}$ anomalies, e.g. the write skew (Figure 7i), and further allows behaviours not allowed under $\texttt{SI}$ such as that in Figure 7h. The kv-store $\mathcal{K}$ is reachable by executing transactions t_1, t_2, t_3 and t_4 in order. In particular, t_4 is executed using $u = \{k_1 \mapsto \{0\}, k_2 \mapsto \{0, 1\}\}$. However, $\mathcal{K}$ is not reachable under $\mathtt{ET}_{\mathtt{SI}}$. This is because t_4 cannot be executed using u under $\mathtt{SI}$: t_4 reads the k_2 version written by t_3; but as $(t_2, t_3) \in \mathsf{RW}$ and $(t_1, t_2) \in \mathsf{WW}$, then u should contain the k_1 version written by t_1, contradicting the fact that t_4 reads the initial version of k_1. The two consistency models are very similar in that many applications that are correct under $\mathtt{SI}$ are also correct under $\mathtt{WSI}$. We give examples of such applications in Section 5.2.

Correctness of $\mathtt{ET}$ Our definitions of consistency models over kv-stores and client views are equivalent to well-known definitions of consistency models over abstract executions [11], and hence over dependency graphs [14]. Given a model M in Figure 6, let $\mathsf{CM}(\mathtt{ET}_M)$ denote the consistency model induced by execution test $\mathtt{ET}_M$ of M. For example, when $M = \mathtt{CC}$, then $\mathsf{CM}(\mathtt{ET}_{\mathtt{CC}})$ denotes the consistency model induced by execution test $\mathtt{ET}_{\mathtt{CC}}$ of $\mathtt{CC}$. Also, let $\mathsf{CM}(\mathcal{A}_M)$ denote the consistency model of M defined on abstract excutions, induced by the set of axioms $\mathcal{A}_M$[11]. For example, when $M = \mathtt{CC}$, then $\mathsf{CM}(\mathcal{A}_{\mathtt{CC}})$ denotes the consistency mode of $\mathtt{CC}$ induced by the $\mathtt{CC}$ axioms on abstract executions.

▶ **Theorem 10.** *For all consistency models M in Figure 6,* $\mathsf{CM}(\mathtt{ET}_M) = \mathsf{CM}(\mathcal{A}_M)$.

The full proof is given in [46], where we define an *intermediate* operational semantics on abstract executions parametrised by axioms, and each step corresponds to an atomic transaction. This is in contrast to [35] which defines a more fine-grained operational semantics.

5 Applications

We use our operational semantics to verify distributed protocols (Section 5.1) and prove invariants of transactional libraries (Section 5.2).

5.1 Application: Verifying Database Protocols

Kv-stores and client 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 [33], satisfies $\mathtt{CC}$. We refer the reader to [46] for the full details. In [46], we also apply the same method to verify that Clock-SI [21], a partitioned database, satisfies$\mathtt{SI}$.

COPS Protocol COPS is a fully replicated database, with each replica storing multiple versions of each key as shown in Figure 8a. Each COPS version ν such as $(k_1, v_1, (t_1, r_1), \emptyset)$ in Figure 8a, 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 the version with time-stamp (t, r) is included in the dependency set of the version with time-stamp (t', r'). COPS assumes a total order over replica identifiers. As such, versions can be totally ordered lexicographically.

The COPS API provides two operations: (1) $\mathtt{put}\,(k, v)$ for writing to a *single* key k; and (2) $\mathtt{read}\,(K)$ for atomically reading from *a set* of keys K. Operations from a client are processed by a single replica. Each client maintains a *context*, which is a set of dependencies tracking the versions the client observes.

$r1$

$(k_1, v_0, (t_0, r_0), \emptyset)$ $(k_2, v_0, (t_0, r_0), \emptyset)$

$(k_1, v_1, (t_1, r_1), \emptyset)$

$r2$

$(k_1, v_0, (t_0, r_0), \emptyset)$ $(k_2, v_0, (t_0, r_0), \emptyset)$

$(k_1, v_1', (t_1, r_2), \emptyset)$ $(k_2, v_2', (t_2, r_2), \{(k_1, t_1, r_2)\})$

(a) Client cl_1 commits a new version of k_1 with 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.

$r1$

$(k_1, v_0, (t_0, r_0), \emptyset)$ $(k_2, v_0, (t_0, r_0), \emptyset)$

$(k_1, v_1, (t_1, r_1), \emptyset)$ $(k_1, v_1', (t_1, r_2), \emptyset)$

$(k_2, v_2', (t_2, r_2), \{(k_1, t_1, r_2)\})$

(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.

$r1$

$(k_1, v_0, (t_0, r_0), \emptyset)$ $(k_2, v_0, (t_0, r_0), \emptyset)$

$(k_1, v_1, (t_1, r_1), \emptyset)$ $(k_1, v_1', (t_1, r_2), \emptyset)$

$(k_2, v_2', (t_2, r_2), \{(k_1, t_1, r_2)\})$

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

Figure 8 COPS protocol

We demonstrate how a COPS client cl interacts with a replica through the following example: $\mathsf{P_{cops}} \triangleq cl : \mathtt{put}\,(k_1, v_1)\,;\ \mathtt{read}\,([k_1, k_2])$. For brevity, we assume that there are two keys, k_1 and k_2, and two replicas, r_1 and r_2, where $r_1 < r_2$ (Figure 8a). 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 Figure 8a. 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.

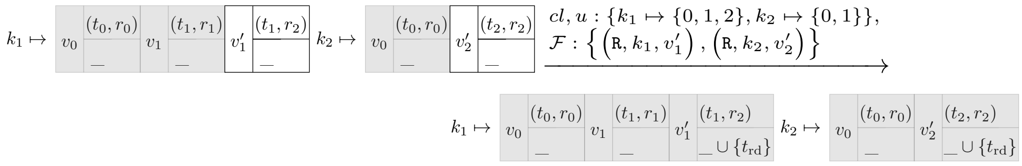
To execute its second multi-read transaction, client cl requests to read from the k_1, k_2 keys by sending the message $\{k_1, k_2\}$ to replica r_1 and awaits a reply. Upon receiving this message, r_1 builds a DEP*-closed snapshot* (a mapping from $\{k_1, k_2\}$ to values) in two phases as follows. 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, Figure 8b 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 Figure 8c, 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 is sent to the client, where it is added to the client context. For their specific setting, Lloyd et al. [33] informally argue that the snapshot sent to the client is causally consistent. By contrast, in what follows we verify the COPS protocol with our general definition of $\mathtt{CC}$.

$$\Theta_0 \xrightarrow{cl,r_1:(\mathtt{W},k_1,(t_1,r_1))} \Theta_1 \xrightarrow{cl,r_1:\mathtt{s}} \Theta_2 \xrightarrow{cl,r_1:(\mathtt{R},k_1,(t_1,r_1))} \Theta_3 \xrightarrow{r_1:\mathtt{sync}} \Theta_4 \xrightarrow{cl,r_1:(\mathtt{R},k_2,(t_2,r_2))}$$
$$\Theta_5 \xrightarrow{cl,r_1:\mathtt{p}} \Theta_6 \xrightarrow{\iota} \Theta_7 \xrightarrow{cl,r_1:(\mathtt{R},k_1,(t_1,r_2))} \Theta_8 \xrightarrow{\iota'} \Theta_9 \xrightarrow{cl,r_1:(\mathtt{R},k_2,(t_2,r_2))} \Theta_{10} \xrightarrow{cl,r_1:\mathtt{e}} \cdots$$

(a) The COPS trace that produces Figures 8b and 8c

$$\Theta'_5 \xrightarrow{\iota} \Theta'_6 \xrightarrow{\iota'} \Theta'_7 \xrightarrow{cl,r_1:\mathtt{p}} \Theta'_8 \xrightarrow{cl,r_1:(\mathtt{R},k_1,(t_1,r_2))} \Theta'_9 \xrightarrow{cl,r_1:(\mathtt{R},k_2,(t_2,r_2))} \Theta'_{10} \xrightarrow{cl,r_1:\mathtt{e}} \cdots$$

(b) The normalised COPS trace



(c) The step encoding the multi-read transaction depicted above: the kv-store before update encodes Figure 8a, and the views (highlighted) encoding of the client contexts before and after the update

Figure 9 COPS traces and trace refinement

COPS Verification We define an operational semantics for the COPS protocol, which uses fine-grained single reads and writes of a key. Using our semantics, we then 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 an equivalent 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 a trace 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, the COPS trace that produces Figures 8b and 8c is depicted in Figure 9a, 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 reads $(\mathtt{R}, k_2, (t_2, r_2))$ from r_1; (6) cl enters the second re-fetch phase of the multi-read transaction ($\mathtt{p}$); (7) an arbitrary step ι interferes; (8) cl re-fetches version $(\mathtt{R}, k_1, (t_1, r_2))$ from r_2 and puts it in the snapshot; (9) an arbitrary step ι' interferes; (10) cl puts the version $(\mathtt{R}, k_2, (t_2, r_2))$ in the snapshot; and (11) cl reads the values in the snapshot and commits the transaction ($\mathtt{e}$).

Recall that a multi-read transaction does not execute atomically in the replica, as captured by multiple read transitions in the trace. For example, steps ι and ι' in Figure 9a interleave the multi-read transaction of cl. Note that the optimistic reads are not observable by the client and thus it suffices to show that the reads from the second re-fetch phase are atomic. To show this, we *normalise* the trace as follows. For each multi-read transaction, we move the reads in the re-fetch phase to the right towards the return step $\mathtt{e}$, so that they are no longer interleaved by others. An example of a normalised trace is given in Figure 9b. In each multi-read transaction, the re-fetch phase can only read a version committed before the $\mathtt{p}$ step. For example, in Figure 9a (top) the multi-read transaction of cl can only read versions in Θ_5 and before. As such, normalising does not alter the returned versions of transactions. After normalisation, transactions in the resulting trace appear to execute atomically.

We next show that a normalised COPS trace can be refined to a trace in our operational semantics. To do this, we encode an abstract COPS state Θ as a configuration in our semantics (Figure 9c). We map all the COPS replicas to a single kv-store. The writer of a version in the kv-store is uniquely determined by the time-stamp of the corresponding

COPS version, while the reader set is given by creating new transaction identifiers for the read-only transactions such as the identifier t_{rd} in Figure 9c. For example, the COPS state in Figure 8a can be encoded as the kv-store depicted in Figure 9c. Since 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 views depicted in Figure 9c.

We finally show that every step in the kv-store trace satisfies $\mathtt{ET}_{\mathtt{CC}}$. Note that existing verification techniques [11, 16] 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. $\mathtt{vShift}_{\mathtt{MR}}$ 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. $\mathtt{vShift}_{\mathtt{RYW}}$ 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 $\mathtt{CC}$ (see [46] for the full details).

5.2 Application: Invariant Properties of Transactional Libraries

With our operational semantics, we are able to prove invariant properties of kv-stores, such as: the robustness of the single counter library against $\mathtt{PSI}$; the robustness of a multi-counter library (Section 2) and the well-known banking library [2] against $\mathtt{SI}$; and the correctness of a lock library against $\mathtt{UA}$ and hence $\mathtt{PSI}$, even though the lock library is not robust for these consistency models. The robustness of the multi-counter and banking library follow from a general proof of the robustness of the so-called $\mathtt{WSI}$-safe libraries against $\mathtt{WSI}$, and hence $\mathtt{SI}$. Our robustness results are the first to be proved for client sessions, in contrast with static analysis techniques for checking robustness [7, 12, 14, 35] that did not support client sessions.

Single-counter Library: Robustness A *transactional library* is a set of transactional operations, e.g. the counter library, $\mathtt{Counter}\,(\mathtt{k}) \triangleq \{\mathtt{Inc}\,(\mathtt{k}), \mathtt{Read}\,(\mathtt{k})\}$, given in Section 2. Client programs of the transactional library can access the underlying kv-store using only the operations of the library. A transactional library is *robust* against an execution test $\mathtt{ET}$ if, for all client programs $\mathtt{P}$ of the library, the kv-stores $\mathcal{K}$ obtained under $\mathtt{ET}$ can also be obtained under $\mathtt{SER}$, i.e. given initial kv-store $\mathcal{K}_0$, initial view environment $\mathcal{U}_0$ and an arbitrary client environment $\mathcal{E}$, for any reachable kv-store $\mathcal{K}$ such that $(\mathcal{K}_0, \mathcal{U}_0, \mathcal{E})\,, \mathtt{P} \Rightarrow^*_{\mathtt{ET}} (\mathcal{K}, \_, \_)\,, \_$, then $\mathcal{K} \in \mathsf{CM}(\mathtt{SER})$. Our robustness results use the following theorem (Theorem 11) that a kv-stores obtained under a trace satisfies serialisability if and only if it contains no cycles.

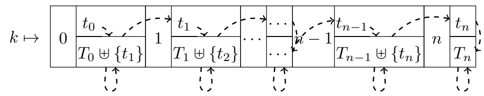
▶ **Theorem 11.** *A kv-store* $\mathcal{K} \in \mathsf{CM}(\mathtt{SER})$ *iff* $(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{RW}_{\mathcal{K}})^+ \cap \mathtt{Id} = \emptyset$.

▶ **Theorem 12.** *The single counter library,* $\mathtt{Counter}\,(\mathtt{k}) \triangleq \{\mathtt{Inc}\,(\mathtt{k}), \mathtt{Read}\,(\mathtt{k})\}$ *given in Section 2, is robust against* $\mathtt{PSI}$.

Proof (sketch). In the single-counter library, $\mathtt{Counter}\,(\mathtt{k})$, a client reads from k by calling $\mathtt{Read}\,(\mathtt{k})$, and writes to k by calling $\mathtt{Inc}\,(\mathtt{k})$ which first reads the value of k and subsequently increments it by one. As $\mathtt{PSI}$ enforces write-conflict freedom ($\mathtt{UA}$), we know that if a transaction t updates k (via $\mathtt{Inc}\,(\mathtt{k})$) 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 $\mathtt{PSI}$ enforces monotonic reads ($\mathtt{MR}$), the order in which clients observe the versions of k (via

$\texttt{Read}\,(\texttt{k})$) is consistent with the order of versions in $\mathcal{K}\,(k)$. As such, the invariant illustrated below always holds (i.e. the kv-store is always has the depicted shape), where $\{t_i\}_{i=1}^{n}$ and $\bigcup_{i=0}^{n} T_i$ denote disjoint sets of transactions calling $\texttt{Inc}\,(\texttt{k})$ and $\texttt{Read}\,(\texttt{k})$, 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 the relation $R \triangleq \mathsf{SO} \cup \{(t, t') \mid \exists i.\, (t{=}t_i \wedge (t'{=}t_{i+1} \vee t' \in T_i)) \vee (t \in T_i \wedge t'{=}t_{i+1})\}$ to a strict total order (i.e. a total, 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. $\texttt{Counter}\,(\texttt{k})$ is robust against $\texttt{PSI}$. By contrast, a multi-counter library on a set of keys K, $\mathsf{Counters}(K) \triangleq \bigcup_{k \in K} \texttt{Counter}\,(\texttt{k})$, is *not* robust against $\texttt{PSI}$. Recall from Section 2 that unlike in $\texttt{SER}$ and $\texttt{SI}$, clients of the multi-counter library under $\texttt{PSI}$ can observe the increments on different keys in different orders (see Figure 7g). Hence, the multi-counter library is not robust against $\texttt{PSI}$. ◀

WSI-safe Libraries: Robustness Our next task is to show that the multi-counter library and the banking library from [2] are robust against $\texttt{SI}$. We do this by defining the notion of $\texttt{WSI}$-safe transactional libraries, and proving a general robustness result for such libraries against $\texttt{WSI}$, and thus $\texttt{SI}$. The proof of this general result uses the following two acyclic properties of kv-stores, where $\texttt{ET}_\top$ is the most permissive execution test (Definition 9).

▶ **Theorem 13.** *Any kv-store* $\mathcal{K} \in \mathsf{CM}(\texttt{ET}_\top)$ *satisfies* $(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}})^+ \cap \texttt{Id} = \emptyset$.

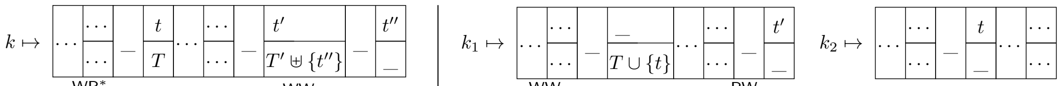
Proof (sketch). From the definition of CM (Definition 9) we know a kv-store $\mathcal{K} \in \mathsf{CM}(\texttt{ET}_\top)$ must be reachable with a given program. This means that Theorem 13 can be seen as an invariant property. We prove it by induction on the length of a trace. For the base case, the initial kv-store $\mathcal{K}_0$ trivially contains no cycles. For the inductive case, since local computation steps do not rely on the kv-store, let us focus on the case where the last transaction step has the form: $(\mathcal{K}, \mathcal{U}, \mathcal{E})\,, \texttt{P} \xrightarrow{(cl, u, \mathcal{F})}_{\texttt{ET}} (\mathcal{K}', \mathcal{U}', \mathcal{E}')\,, \texttt{P}'$, where $\mathcal{K}$ contains no $R \triangleq (\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}})$ cycles by the inductive hypothesis. Let t be the new transaction in $\mathcal{K}'$. We then proceed by contradiction and assume that $\mathcal{K}'$ has a R cycle. As $\mathcal{K}$ contains no R cycles, this cycle must involve t, i.e. $t \xrightarrow{R} t_1 \xrightarrow{R} \cdots \xrightarrow{R} t_n \xrightarrow{R} t$, where $t_1, \cdots, t_n$ are distinct. As t is the last transaction and $t \notin \mathcal{K}$, we cannot have $t \xrightarrow{\mathsf{SO}} t_1$. Similarly, all versions written by t have empty reader sets, and . thus we cannot have $t \xrightarrow{\mathsf{WR}_{\mathcal{K}'}} t_1$. This then leads to a contradiction as $t \xrightarrow{\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}'}} t_1$. Therefore, the new kv-store $\mathcal{K}'$ satisfies $(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}'})^+ \cap \texttt{Id} = \emptyset$. ◀

▶ **Theorem 14.** *Any kv-store* $\mathcal{K} \in \mathsf{CM}(\texttt{ET}_{\texttt{CP}})$ *satisfies* $((\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}); \mathsf{RW}_{\mathcal{K}}^?)^+ \cap \texttt{Id} = \emptyset$.

Proof (sketch). We proceed as in the proof of Theorem 13. For the inductive case, consider $(\mathcal{K}, \mathcal{U}, \mathcal{E})\,, \texttt{P} \xrightarrow{(cl, u, \mathcal{F})}_{\texttt{ET}} (\mathcal{K}', \mathcal{U}', \mathcal{E}')\,, \texttt{P}'$, where $\mathcal{K}$ contains no $R \triangleq ((\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}); \mathsf{RW}_{\mathcal{K}}^?)$ cycles by the inductive hypothesis. Let us then assume $\mathcal{K}'$ has a R cycle which must include the new transaction t. There are then two cases as follows where $t_1, \cdots, t_n$ are distinct:

(1) $t \xrightarrow{R} t_1 \xrightarrow{R} \cdots \xrightarrow{R} t_n \xrightarrow{R} t$

This cycle cannot exist as t is the last transaction in $\mathcal{K}'$. More concretely, as in Theorem 13 we know we cannot have $t \xrightarrow{\mathsf{SO}} t_1$ or $t \xrightarrow{\mathsf{WR}_{\mathcal{K}'}} t_1$. For analogous reasons, we cannot have $t \xrightarrow{\mathsf{SO}} t' \xrightarrow{\mathsf{RW}_{\mathcal{K}'}} t_1$ or $t \xrightarrow{\mathsf{WR}_{\mathcal{K}'}} t' \xrightarrow{\mathsf{RW}_{\mathcal{K}'}} t_1$, for some transaction $t' \in \mathcal{K}$.



(a) $t \xrightarrow{\mathsf{WR}^*} t''$ replaces $t \xrightarrow{\mathsf{WW}} t''$ **(b)** $t \xrightarrow{\mathsf{WW}} t'$ replaces $t \xrightarrow{\mathsf{RW}} t'$ where t has a write

Figure 10 `WSI`-safety

(2) $t_1 \xrightarrow{R} \cdots \xrightarrow{R} t_n \xrightarrow{(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}'})} t \xrightarrow{\mathsf{RW}_{\mathcal{K}'}} t_1$
From $\mathtt{ET_{CP}}$ the view u of t must contains all versions written by $t_1, \cdots, t_n$. As such, we cannot have $t \xrightarrow{\mathsf{RW}_{\mathcal{K}'}} t_1$ as by $\mathsf{RW}_{\mathcal{K}'}$ we know u is behind the versions written by t_1.

Specific libraries [2, 5, 7] have been shown to be robust against `SI` by individually checking all final results of all their client programs. By contrast, we identify the notion of a `WSI`-*safe* library and prove that such a library is robust against `WSI`, and hence `SI`, by showing that the acyclic invariant given in Theorem 11 is preserved by each transition step.

▶ **Definition 15** (`WSI`-safe). *A library is* `WSI`-safe *if and only if, for all its client programs* `P` *and all kv-stores* $\mathcal{K}$*, if* $\mathcal{K}$ *is obtained by executing* `P` *under* `WSI`[7]*, then for all* t, k, i, i'*:*

$$t \in \mathsf{rs}\left(\mathcal{K}\left(k,i\right)\right) \wedge t \neq \mathsf{w}\left(\mathcal{K}\left(k,i'\right)\right) \Rightarrow \forall k', j.\, t \neq \mathsf{w}\left(\mathcal{K}\left(k',j\right)\right), \tag{1}$$
$$t \neq t_0 \wedge t = \mathsf{w}\left(\mathcal{K}\left(k,i\right)\right) \Rightarrow \exists j.\, t \in \mathsf{rs}\left(\mathcal{K}\left(k,j\right)\right), \tag{2}$$
$$t \neq t_0 \wedge t = \mathsf{w}\left(\mathcal{K}\left(k,i\right)\right) \wedge \exists k', j, j'.\, t \in \mathsf{rs}\left(\mathcal{K}\left(k',j\right)\right) \Rightarrow t = \mathsf{w}\left(\mathcal{K}\left(k',j'\right)\right). \tag{3}$$

That is, (1) if a transaction t reads from k but does not write to it, then t must be a read-only transaction; (2) if t writes to k, then it must also read from it, a property known as *no-blind writes*[8]; and (3) if t writes to k, then it must also write to all keys it reads from. The read-only transactions, satisfying (1), can be reordered to be next to the write that they are reading. Their behaviour is, thus, serialisable in that the write they are reading is current. Under `WSI` and `SI`, transactions satisfying *strict no-blind writes* (i.e. (2) and (3)) enforce a total order over transactions on a key, which is enough to obtain serialisable behaviour.

It is straightforward to see that the multi-counter library given in Section 2 is `WSI`-safe; we will show that the banking example in [2] is `WSI`-safe. The example in [7] is `WSI`-safe. In [5], there are many examples of libraries that are shown to be robust against `SI`: the smaller examples are `WSI`-safe; the larger examples have not been checked.

▶ **Theorem 16** (`WSI` robustness). *A* `WSI`*-safe library is robust against* `WSI`*.*

Proof (sketch). Pick a `WSI`-safe library L, a client program `P` of L and a kv-store $\mathcal{K}$ obtained from executing `P` under `WSI`, i.e. $\left(\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}\right), \mathtt{P} \Rightarrow^*_{\mathtt{ET_{WSI}}} \left(\mathcal{K}, \_, \_\right), \_$. From Theorem 11 it suffices to prove that $(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{RW}_{\mathcal{K}})^+$ is acyclic. We proceed by contradiction. Let us assume there exists t_1 such that $t_1 \xrightarrow{(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{RW}_{\mathcal{K}})^+} t_1$. From Theorem 13 we know $(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}})^+$ is acyclic. Moreover, thanks to no-blind-writes in (2) and `UA`, any $\mathsf{WW}_{\mathcal{K}}\left(k\right)$ edge on a key k can be replaced by $\mathsf{WR}^+_{\mathcal{K}}(k)$), as illustrated in Figure 10a. As such, $(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}})^+ \cup \mathsf{WW}_{\mathcal{K}}$ is acyclic and thus this cycle is of the form: $t_1 \xrightarrow{R^*} \xrightarrow{\mathsf{RW}} \xrightarrow{R^*} \cdots \xrightarrow{R^*} \xrightarrow{\mathsf{RW}} \xrightarrow{R^*} t_1$, where $R \triangleq \mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}$. From (3) we know an $\mathsf{RW}_{\mathcal{K}}\left(k_1\right)$ edge on a key k_1 starting from a writing transaction t can be replaced by a WW edge, as illustrated in Figure 10b. Moreover, from (2) we know we can replace WW edges by WR^+. We thus have:

[7] That is, for initial kv-store $\mathcal{K}_0$, initial view environment $\mathcal{U}_0$ and arbitrary client environment $\mathcal{E}$, $\left(\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}\right), \mathtt{P} \Rightarrow^*_{\mathtt{ET_{WSI}}} \left(\mathcal{K}, \_, \_\right), \_)$.
[8] From `UA`, it is immediate that $j = i - 1$.

$t_1 \xrightarrow{R'^*}\xrightarrow{\mathsf{RW}}\xrightarrow{R'^+}\cdots\xrightarrow{R'^+}\xrightarrow{\mathsf{RW}}\xrightarrow{R'^*} t_1$, where $R' \triangleq \mathsf{SO} \cup \mathsf{WR}$, i.e. $t_1 \xrightarrow{(R';\mathsf{RW}^?)^*} t_1$. This, however, leads to a contradiction by Theorem 14.

Using Theorem 16, we can prove the robustness of the banking library in [2] against `WSI`, and hence `SI`. Alomari et al. [2] informally showed that this example is robust: they identified a notion of dangerous dependency between transactions which, they argued, can lead to violation of robustness of `SI`; and they argued that this banking example contains no such dangerous dependencies. The original banking example worked with a relational database with three tables *account*, *saving* and *checking*. The account table maps customer names to customer IDs ($\texttt{Account}(\underline{\texttt{Name}}, \texttt{CID})$); the saving table maps customer IDs to their saving balances ($\texttt{Saving}(\underline{\texttt{CID}}, \texttt{Balance})$); and the checking table maps customer IDs to their checking balances ($\texttt{Checking}(\underline{\texttt{CID}}, \texttt{Balance})$). The balance of a saving account must be non-negative, but a checking account may have a negative balance.

For simplicity, we encode the saving and checking tables as a single kv-store, and omit 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 the 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 $n_s \triangleq 2n{+}1$, i.e. $\textsc{Key} \triangleq \bigcup_{n\in\mathbb{N}} \{n_c, n_s\}$. Moreover, if n identifies a customer with $(\_, n) \in \texttt{Account}(\underline{\texttt{Name}}, \texttt{CID})$, 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:

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

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

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

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

$$\texttt{writeCheck}(\texttt{n},\texttt{v}) \triangleq \begin{bmatrix} \texttt{x} := [\texttt{n}_s]\,;\ \texttt{y} := [\texttt{n}_c]\,; \\ \texttt{if}(\texttt{v} > 0\ \&\&\ \texttt{x} + \texttt{y} < \texttt{v})\{\ [\texttt{n}_c] := \texttt{y} - \texttt{v} - 1\ \} \\ \texttt{else}\{\ [\texttt{n}_c] := \texttt{y} - \texttt{v}\ \} \qquad [\texttt{n}_s] := \texttt{x} \end{bmatrix}$$

The $\texttt{balance}(\texttt{n})$ operation returns the total balance of customer `n` in `ret`. The `depositCheck` $(\texttt{n}, \texttt{v})$ deposits `v` to the checking account of customer `n` when `v` is non-negative, otherwise it leaves the checking account unchanged. When $\texttt{v} \geq 0$, $\texttt{transactSaving}(\texttt{n}, \texttt{v})$ deposits `v` to the saving account of `n`. When $\texttt{v} < 0$, $\texttt{transactSaving}(\texttt{n}, \texttt{v})$ withdraws `v` from the saving account of `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 balance of costumer `n` to the checking account of customer $\texttt{n}'$. Lastly, $\texttt{writeCheck}(\texttt{n}, \texttt{v})$ cashes a cheque of customer `n` in the amount `v` by deducting `v` from its checking account. If `n` does not hold sufficient funds (i.e. the combined checking and saving balance is less than `v`), customer `n` is penalised by deducting one additional pound. In [2], the authors argue that to make this library robust against `SI`, the $\texttt{writeCheck}(\texttt{n}, \texttt{v})$ operation must be strengthened by writing back the saving account balance (via $[\texttt{n}_s] := \texttt{x}$), even though this is unchanged.

The banking library is more complex than the multi-counter library. Nevertheless, all banking transactions are either read-only or satisfy the no-blind writes property. Hence, the banking library is `WSI`-safe, and so robust against `WSI` and `SI`.

Lock Library: Mutual-exclusion Guarantee Finally, we demonstrate that, although a distributed lock library is not robust against `UA`, we can nevertheless prove an invariant property stating that only one client can hold the lock at a given time, thus establishing a mutual exclusion guarantee. The distributed lock library provides the following operations on a key k:

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

$$\texttt{lock}\,(k) \triangleq \texttt{do}\{\ \texttt{tryLock}\,(k)\ \}\texttt{until}(\texttt{m}{=}\texttt{false}) \qquad \texttt{unlock}\,(k) \triangleq [[k] := 0]$$

The `tryLock` operation reads the k value; if the value is zero (i.e. the lock is available), then it sets it to the client ID and returns `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 k value to zero.

Consider the program $\mathtt{P_{LK}}$ where clients cl and cl' compete to acquire the lock k:

$$\mathtt{P_{LK}} \triangleq (cl : (\texttt{lock}\,(k)\,;\ \ldots;\ \texttt{unlock}\,(k))^* \parallel cl' : (\texttt{lock}\,(k)\,;\ \ldots;\ \texttt{unlock}\,(k))^*)$$

The locking program in $\mathtt{P_{LK}}$ 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, $\mathtt{P_{LK}}$ is robust and hence correct under `WSI` as well as stronger models such as `SI`. However, $\mathtt{P_{LK}}$ is not robust under `UA` or `PSI`: `lock` may read an old value of key k until it reads its most up-to-date value and acquires it. Nevertheless, we show that $\mathtt{P_{LK}}$ is correct under `UA` (and hence `PSI`) in that it satisfies a mutual exclusion guarantee where only one client can hold the lock at a time. We capture this guarantee by the following invariant, stating that for all positive i $(i > 0)$:

$$\mathsf{val}(\mathcal{K}\,(k,i)) \neq 0 \Leftrightarrow \mathsf{val}(\mathcal{K}\,(k,i-1)) = 0 \tag{4}$$

$$\mathsf{val}(\mathcal{K}\,(k,i)) = 0 \Rightarrow \mathsf{w}(\mathcal{K}\,(k,i)) = \mathsf{w}(\mathcal{K}\,(k,i-1)) \tag{5}$$

It is straightforward to show that, under `UA`, only one client can hold the lock (4), and the same client releases the lock (5). Assume a kv-store $\mathcal{K}$ satisfies this invariant. Given the lock program in $\mathtt{P_{LK}}$, if the latest value of k is 0, then all clients are competing to acquire k, and thanks to `UA` only a client cl with full view of k can install a new version with its unique client ID. This will stop other clients from acquiring k as the latest value is now non-zero. Subsequently, when cl executes its next transaction, i.e. `unlock` (k), it releases the lock and installs a new version with value zero.

Invariants vs. Execution Graphs We have demonstrated how invariant properties of transactional libraries can be used to prove their robustness, as well as library-specific guarantees such as mutual exclusion. Although existing work can establish the robustness of a library using execution graphs (e.g. dependency graphs of [1]), they typically do this by checking the *final* results of all its client programs. By contrast, thanks to our operational model, we achieve this by establishing an invariant property at each execution step, thus allowing a simpler, more compositional proof. Moreover, whilst it is straightforward for us to prove library-specific guarantees (e.g. mutual exclusion for locks) by simply encoding them as an invariant of the library, establishing such properties using execution graphs is much more difficult. This is because execution graphs do not directly record the library *state* and merely record the execution shape, thus making it harder to reason about such guarantees.

6 Conclusions and Future Work

We have introduced an interleaving operational semantics for describing the client-observable behaviour of atomic transactions over distributed kv-stores, using abstract states comprising global, centralised kv-stores, partial client views, and transition steps parametrised by an execution test which directly captures when a transaction is able to commit on a state. Using these execution tests, we provide a general definition of consistency model and provide example instantiations including CC, PSI, SI and SER. In [46], we prove that our definitions are equivalent to the existing definitions in the literature that use execution graphs [11].

We have used our semantics to verify that protocols of real-world distributed databases satisfy particular consistency models, e.g. that the replicated database COPS [33] satisfies CC, and the partitioned database Clock-SI [21] satisfies SI. These results contrast with those of [21, 33], which justify the correctness of implementations using consistency model definitions that are specific to the implementations. We have also proved several invariant properties for clients, showing that the clients of several libraries (single-counter, multi-counter and banking libraries) are robust against the appropriate models, and showing that certain clients of a lock library satisfy a mutual exclusion property under PSI, even though they are not robust against PSI. We thus believe that our semantics provides an interesting abstract interface between distributed implementations and clients. We plan to validate further the usefulness of our semantics by verifying other well-known protocols of distributed databases [4, 30, 34, 43], exploring robustness results for OLTP workloads such as TPC-C [44] and RUBiS [39], and exploring other program analysis techniques such as transaction chopping [13, 41], invariant checking [24, 47] and program logics [27]. We also plan to develop tools to generate litmus tests for implementations and to analyse client programs.

Our work assumes the *snapshot property* and the *last-write-wins* policy, common assumptions in real-world distributed databases. Under these assumptions, we are not aware of a consistency model that we cannot express using our semantics. There are consistency models that do not satisfy these assumptions, e.g. *read committed* [4] captured in [16]. In future, we will explore whether it is possible to weaken our assumptions to express such weak consistency models. This might be possible by introducing 'promises' in the style of [28].

There are many resonances between the high-level behaviour of distributed systems and the low-level behaviour of weak memory. Indeed, our partial client views were inspired by the views of the 'promising' C11 semantics in [28]. In future, we plan to explore whether our semantics of atomic transactions can be loosened to describe the more fine-grained behaviour of transactions on weak memory [38, 15]. We are also interested in the work of Doherty et al. [20], describing an operational semantics and a program logic for the release-acquire (RA) fragment of C11, which, interestingly, is based on dependency graphs. We believe that we can adapt our semantics to model the RA fragment, using simple read-write primitives rather than atomic transactions and a variant of our definition of causal consistency.

References

1. Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1999. URL: http://pmg.csail.mit.edu/papers/adya-phd.pdf.
2. M. Alomari, M. Cahill, A. Fekete, and U. Rohm. The cost of serializability on platforms that use snapshot isolation. In *2008 IEEE 24th International Conference on Data Engineering*, pages 576–585, April 2008. doi:10.1109/ICDE.2008.4497466.
3. Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *Proceedings of the 2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, SRDS '13, page 163–172, USA, 2013. IEEE Computer Society. doi:10.1109/SRDS.2013.25.

4 Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Scalable atomic visibility with ramp transactions. *ACM Trans. Database Syst.*, 41(3), July 2016. doi:10.1145/2909870.

5 Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. Checking robustness against snapshot isolation. *CoRR*, abs/1905.08406, 2019. URL: http://arxiv.org/abs/1905.08406, arXiv:1905.08406.

6 Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD'95, pages 1–10. ACM, 1995. doi:10.1145/223784.223785.

7 Giovanni Bernardi and Alexey Gotsman. Robustness against Consistency Models with Atomic Visibility. In *Proceedings of the 27th International Conference on Concurrency Theory*, pages 7:1–7:15, 2016. doi:10.4230/LIPIcs.CONCUR.2016.7.

8 Carsten Binnig, Stefan Hildenbrand, Franz Färber, Donald Kossmann, Juchang Lee, and Norman May. Distributed Snapshot Isolation: Global Transactions Pay Globally, Local Transactions Pay Locally. *The VLDB Journal*, 23(6):987–1011, December 2014. doi:10.1007/s00778-014-0359-9.

9 Sebastian Burckhardt, Manuel Fahndrich, Daan Leijen, and Mooly Sagiv. Eventually Consistent Transactions. In *Proceedings of the 21nd European Symposium on Programming*. Springer, March 2012. doi:10.1007/978-3-642-28869-2_4.

10 Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 568–590, 2015. doi:10.4230/LIPIcs.ECOOP.2015.568.

11 Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A Framework for Transactional Consistency Models with Atomic Visibility. In Luca Aceto and David de Frutos Escrig, editors, *26th International Conference on Concurrency Theory (CONCUR 2015)*, volume 42 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 58–71, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2015/5375, doi:10.4230/LIPIcs.CONCUR.2015.58.

12 Andrea Cerone and Alexey Gotsman. Analysing Snapshot Isolation. In *Proceedings of the 2016 ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC'16, pages 55–64. ACM, 2016. doi:10.1145/2933057.2933096.

13 Andrea Cerone, Alexey Gotsman, and Hongseok Yang. Transaction chopping for parallel snapshot isolation. In *Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363*, DISC 2015, page 388–404, Berlin, Heidelberg, 2015. Springer-Verlag. doi:10.1007/978-3-662-48653-5_26.

14 Andrea Cerone, Alexey Gotsman, and Hongseok Yang. Algebraic Laws for Weak Consistency. In Roland Meyer and Uwe Nestmann, editors, *Proceedings of the 27th International Conference on Concurrency Theory*, volume 85 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:18, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CONCUR.2017.26.

15 Nathan Chong, Tyler Sorensen, and John Wickerson. The semantics of transactions and weak memory in x86, power, arm, and C++. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 211–225. ACM, 2018. doi:10.1145/3192366.3192373.

16 Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *Proceedings of the 2017 ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC'17, pages 73–82, New York, NY, USA, 2017. ACM. doi:10.1145/3087801.3087802.

17 Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A Logic for Time and Data Abstraction. In Richard E. Jones, editor, *Proceedings of the 28th European Conference on Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 207–231. Springer, July 2014. doi:10.1007/978-3-662-44202-9_9.

18 Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, page 715–726. VLDB Endowment, 2006. doi:10.5555/1182635.1164189.

19 Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent Abstract Predicates. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP'10, pages 504–528. Springer-Verlag, 2010. doi:10.1007/978-3-642-14107-2_24.

20 Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. Verifying C11 programs operationally. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 355–365, New York, NY, USA, 2019. ACM. doi:10.1145/3293883.3295702.

21 Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-SI: Snapshot Isolation for Partitioned Data Stores Using Loosely Synchronized Clocks. In *Proceedings of the 32nd Leibniz International Proceedings in Informatics (LIPIcs)*, SRDS'13, pages 173–184, Washington, DC, USA, 2013. IEEE Computer Society. doi:10.1109/SRDS.2013.26.

22 Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Database replication using generalized snapshot isolation. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, SRDS '05, page 73–84, USA, 2005. IEEE Computer Society. doi:10.1109/RELDIS.2005.14.

23 Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making Snapshot Isolation Serializable. *ACM Transactions on Database Systems*, 30(2):492–528, June 2005. doi:10.1145/1071610.1071615.

24 Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 371–384, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2837614.2837625.

25 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 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*, POPL'15, pages 637–650. ACM, 2015. doi:10.1145/2676726.2676980.

26 Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. Safe replication through bounded concurrency verification. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276534.

27 Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. Alone Together: Compositional Reasoning and Inference for Weak Isolation. *Proceedings of the ACM on Programming Languages*, 2(POPL):27:1–27:34, December 2017. doi:10.1145/3158115.

28 Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A Promising Semantics for Relaxed-memory Concurrency. In *Proceedings of the 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'17, pages 175–189, New York, NY, USA, 2017. ACM. URL: http://doi.acm.org/10.1145/3009837.3009850, doi:10.1145/3009837.3009850.

29 Eric Koskinen and Matthew Parkinson. The push/pull model of transactions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 186–195, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2737924.2737995.

30 Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 265–278, USA, 2012. USENIX Association. URL: `http://www.cs.otago.ac.nz/cosc440/readings/osdi12-final-162.pdf`.

31 Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, December 1975. `doi:10.1145/361227.361234`.

32 Si Liu, Peter Csaba Ölveczky, Keshav Santhanam, Qi Wang, Indranil Gupta, and José Meseguer. ROLA: A New Distributed Transaction Protocol and Its Formal Analysis. In Alessandra Russo and Andy Schürr, editors, *Fundamental Approaches to Software Engineering*, pages 77–93, Cham, 2018. Springer. `doi:10.1007/978-3-319-89363-1_5`.

33 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 401–416, New York, NY, USA, 2011. Association for Computing Machinery. `doi:10.1145/2043556.2043593`.

34 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *Presented as part of the 10^{th} USENIX Symposium on Networked Systems Design and Implementation*, pages 313–328, Lombard, IL, 2013. USENIX. URL: `https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/lloyd`.

35 Kartik Nagar and Suresh Jagannathan. Automated Detection of Serializability Violations Under Weak Consistency. In *Proceedings of the 29^{th} International Conference on Concurrency Theory*, pages 41:1–41:18, 2018. `doi:10.4230/LIPIcs.CONCUR.2018.41`.

36 Aleksandar Nanevski, Yuy Ley-wild, Ilya Sergey, and Germán Andrés Delbianco. *Communicating State Transition Systems for fine-grained concurrent resources*, pages 290–310. Lecture Notes in Computer Science. springer-verlag, 2014. `doi:10.1007/978-3-642-54833-8_16`.

37 Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, October 1979. `doi:10.1145/322154.322158`.

38 Azalea Raad, Ori Lahav, and Viktor Vafeiadis. On Parallel Snapshot Isolation and Release/Acquire Consistency. In Amal Ahmed, editor, *Proceedings of the 27^{th} European Symposium on Programming*, pages 940–967, Cham, 2018. Lecture Notes in Computer Science. `doi:10.1007/978-3-319-89884-1_33`.

39 The RUBiS benchmark, 2008. URL: `https://rubis.ow2.org/index.html`.

40 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, page 386–400, Berlin, Heidelberg, 2011. Springer-Verlag. `doi:10.1007/978-3-642-24550-3_29`.

41 Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction Chopping: Algorithms and Performance Studies. *ACM Transactions on Database Systems*, 20(3):325–363, September 1995. URL: `http://doi.acm.org/10.1145/211414.211427`, `doi:10.1145/211414.211427`.

42 Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional Storage for Geo-replicated Systems. In *Proceedings of the 23^{rd} ACM Symposium on Operating Systems Principles*, SOSP'11, pages 385–400, New York, NY, USA, 2011. ACM. URL: `http://doi.acm.org/10.1145/2043556.2043592`, `doi:10.1145/2043556.2043592`.

43 Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. Wren: Nonblocking Reads in a Partitioned Transactional Causally Consistent Data Store. In *Proceedings of the 48^{th} Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN'18, pages 1–12, 2018. `doi:10.1109/DSN.2018.00014`.

44 The TPC-C benchmark, 1992. URL: `http://www.tpc.org/tpcc/`.

45 Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009. `doi:10.1145/1435417.1435432`.

46 Shale Xiong. *Parametric Operational Semantics for Consistency Models*. PhD thesis, Imperial College London, April 2021. URL: `http://www.shalexiong.com/thesis.pdf`.

47 Peter Zeller. Testing properties of weakly consistent programs with repliss. In *Proceedings of the 3^{rd} International Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC'17, pages 3:1–3:5, New York, NY, USA, 2017. ACM. URL: `http://doi.acm.org/10.1145/3064889.3064893`, `doi:10.1145/3064889.3064893`.