Imperial College of Science, Technology and Medicine
Department of Computing

Parametric Operational Semantics for Consistency Models

Shale Xiong

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London and
the Diploma of Imperial College London, October 2019

Abstract

Cloud computing has become popular for its low cost. A storage sub-system is a key component in many cloud computing infrastructures, and many systems have used so-called "NoSQL" databases, where data is often organised in a key-value structure, for example Dynamo DB, a distributed key-value store from Amazon Web Service (AWS). This is driven by the need to store unstructured data, such as pictures, videos, or documents. Similar to traditional relational databases, transactions are the de facto interfaces in cloud storages. Many distributed cloud storages often provide high availability and fault-tolerance, but adopt *weak consistency*, where individual server is allowed to operate without synchronisation in certain situation. Engineers and researchers have proposed various weak consistency models via reference implementations in their specific setting. However, there has been little work on formal, implementation-independent definitions of consistency models. We introduce an interleaving operational semantics, with the focus on the client-observable behaviour of atomic transactions on distributed key-value stores. Our semantics builds on abstract states comprising centralised, global key-value stores, representing the overall states of distributed systems and multiple, mutually independent, partial client views, representing client-observable states. In each step, a client with its view commits a transaction to the abstract key-value store, and this step must satisfy certain conditions of the chosen consistency model, called an execution test, which is a novel operational definition of this consistency model. We provide definitions of various well-known consistency models such as snapshot isolation and causal consistency and show that our definitions are equivalent to the well-known declarative definitions of consistency models. We then explore two immediate applications of our semantics: specific implementation protocols can be verified in our operational semantics via trace refinement; client programs can be shown to satisfies invariant properties. These two applications show that our operational semantics captures the interfaces between client programs and implementation protocols.

Dedication

I herewith certify that all material in this thesis which is not my own work has been properly acknowledged.

Shale Xiong

Copyright

‘Explore wide, yet garner with caution; be erudite, yet comment with consideration.’

Shi Su

“博观而约取，厚积而薄发”

苏轼

Acknowledgements

It is my honour to work with my supervisor, Philippa Gardner, my closest colleagues, Andrea Cerone and Azalea Raad, and the entire research group. They kindly provided me with help and guidance.

I also want to spatially thank Oana Cocarascu. We shared the office for 4 years and it was joyful 4 years. Without her, I would have not known how to enjoy the unfamiliar city as an international student.

Lastly, thanks to my parents for their support. They provided me the chance to study aboard.

Contents

List of Figures

List of Definitions and Proofs

Chapter 1

Introduction

The internet is an essential part of modern society. On the internet, millions of applications provide various services to improve life standard. People from all over the world can access these services by simply a few clicks. However, behind the hood, the applications need to incorporate massive client requests globally. This means that the cost of such globally scaled applications is very high. Thus, a few big companies, that have already built and maintained global storage and connectivity infrastructure for their internal applications, have stared renting out their infrastructure so that other applications can focus on their businesses without worrying too much about the low-level functionalities. People often refer to such infrastructure as *clouds*: for example, Amazon Web Service (AWS) [Amazon, 2019] and Google Cloud Platform (GCP) [Google, 2019]. They have become very popular for their low cost, high availability and high fault-tolerance. Applications that are built on such clouds treat the clouds as *black boxes* with which to interact via fixed interface. Yet behind the interface, the always-on and high-performance infrastructure is implemented via globally-scaled *distributed systems*. There are hundreds of data centres globally, each of which hosts tens of thousands of servers. All these data centres are connected with low leniency lines, over which a well-designed and well-engineered protocol guarantees consistency between centres.

One important service of the clouds are data storages or *databases*, since many applications need to constantly access data. Transactions are the *de facto* interface for databases. In contrast to traditional centralised relational databases with transactions consisting of SQL queries, many clouds focus on key-value stores with simple transaction interface. This means that each interface call needs less computation to increase the throughput and decrease the response time. One example of a key-value store is Amazon DynamoDB [Amazon, 2019; DeCandia et al., 2007], a key-value database in AWS. Amazon DynamoDB provides `GetItem`, `PutItem` and `DeleteItem` operations per key, and `TransactGetItems` and `TransactWritesItems` for reading and writing a list of keys, respectively.

In traditional centralised databases, transactions can be seen as if they are executed one-by-one: this is known as *strict serialisibility*, the strongest *consistency model*. The implementations of

strict serialisibility require that transactions appear to be executed in some global order, however, these transactions can be executed in a fine-grained way. System designers, then, proposed *weak consistency models*, which allows more fine-grained interleaving. For example, the three SQL isolation level, *read uncommitted*, *read committed* and *repeatable read* allows different level of fine-grained interleaving between transactions. In distributed databases, because of the CAP theorem [Gilbert and Lynch, 2002], to achieve high availability and high fault-tolerance, they *must* sacrifice strong consistency and instead use *weak consistency models*. Engineers and researchers for centralised and distributed databases have proposed many protocols employing weak consistency models, with the focus on designing the internal implementation strategies to tackle real-world constraints and maximise performance [Bailis et al., 2014; Lloyd et al., 2011; Du et al., 2013; Lloyd et al., 2013; DeCandia et al., 2007]. The semantics of many *specific consistency models* is originally captured by specific reference implementations. For example, Berenson et al. [1995] proposed the implementation strategy for *snapshot isolation* (`SI`) on centralised databases. In the implementation, each transaction works on a snapshot of the database taken at the beginning, and commits the effect at the end. Many works [Daudjee and Salem, 2006; Elnikety et al., 2005; Du et al., 2013] generalised this idea in distributed databases. However, in distributed databases, data are distributed in servers, and synchronisation messages between servers might be delayed or dropped due to internet malfunction. Implementer proposed *parallel snapshot isolation* (`PSI`) [Sovran et al., 2011; Raad et al., 2018; Ardekani et al., 2013], where a transaction only takes a snapshot of a simple server. *Causal consistency* (`CC`) introduces causal relation over transactions [Hutto and Ahamad, 1990; Lloyd et al., 2011; Petersen et al., 1997; Belaramani et al., 2006] This model requires if a transaction observes the effect of another transaction t, it must observe all transactions t depends on. *Eventual consistency* is the base-line for many distributed systems, for example, Amazon DynamoDB [DeCandia et al., 2007; Bailis et al., 2014], where transactions are always allowed to read from a site without any synchronisation, but all synchronisation message must eventually be delivered to all sites.

However, the details of such implementations are irrelevant for the clients; the clients can only interact with these systems via the transaction interface. The behaviour of the transaction interface is subtle under weak consistency models since that transactions are allowed to read old values subject to certain constraints. Thus, these weak behaviours pose several challenges: the formalisation of client-observable behaviour and the verification of database protocols and client applications.

Much work has been done to formalise the semantics of such weak consistency models using declarative semantics. Several *graph-based general* formalisms have been proposed, such as dependency graphs [Adya, 1999] and abstract executions [Burckhardt et al., 2012; Cerone et al., 2015a], to provide a unified semantics for formulating different consistency models. Dependency graphs are directed graphs where nodes represent transactions and edges represent the dependency between transactions including write-read (WR), write-write (WW) and read-write

(RW) dependency relations. Adya [1999] introduced dependency graphs to formalise SQL isolation levels. Later, they are used to model weak memory behaviours [Doherty et al., 2019; Alglave, 2010; Batty et al., 2016; Lahav et al., 2017, 2016; Batty et al., 2011; Chong et al., 2018; Batty, 2014; Dongol et al., 2018]. Since the dependency relations can be over-approximated statically from the code, many program analysis techniques on transactional applications are built over dependency graphs [Fekete et al., 2005; Bernardi and Gotsman, 2016; Beillahi et al., 2019; Cerone and Gotsman, 2016]. However, it is not easy to verify implementation protocols for distributed systems using dependency graphs, because the three relations in dependency graphs do not have direct connection to well-known implementation strategies.

Abstract executions are also directed graphs where nodes represent transactions and edges are labelled with arbitration relation (AR) and visibility relation (VIS). The arbitration relation is a total order over transactions. In centralised databases, this relation is the commit order. In distributed databases, a *resolution policy* determines the order between transactions that write to the same key, which overall is a partial order over all transactions. The arbitration relation, therefore, can be recovered from the partial order. However, a transaction may not observe all transactions that commit before, since transactions may commit to different sites. The visibility relation determines, for every transaction, the set of visible transactions, which then determines the actual read value for each key. Distributed database engineers and researchers prefer to use abstract executions to define consistency models on distributed systems [Burckhardt et al., 2012; Cerone and Gotsman, 2016; Cerone et al., 2015a] and replicated data types [Burckhardt et al., 2014]. With the help of the total arbitration relation, these definitions of consistency models match the English descriptions given by the implementers. Moreover, abstract executions are useful for verifying implementation protocols. However, it is not clear how to directly use abstract executions to reason about client programs, which requires client-observable history.

These two graph-based declarative models are well studied and both of them can cover many consistency models. Cerone et al. [2017] have shown the connection between dependency graphs and abstract executions. They convert definitions of consistency models on abstract executions in certain patterns to those on dependency graphs and vice versa. However, these two graphs define consistency models in an axiomatic style in the sense that they require the resulting graphs obtained by executing the whole programs to the end and then *rule out* the invalid graphs against the axioms associated with the shape of graphs. This is different from operational semantics which describes valid states and how they evolve. This means that axiomatic approaches cannot be used to reason about invariants of client programs. Additionally, compared with dependency graphs, abstract executions are better for verifying implementation protocols. However, it is not an easy task to verify implementation protocols, since many protocols for *specific* consistency models are captured using reference implementations [Berenson et al., 1995; Sovran et al., 2011; Raad et al., 2018; Bailis et al., 2014; Du et al., 2013; Lloyd et al., 2011; Petersen et al., 1997], that is, in an operational style.

Unlike declarative approaches, there has been little work on *general* operational semantics for

describing a range of consistency models. Nagar and Jagannathan [2018] proposed a graph-based operational semantics, which constructs abstract executions step by step. They develop a model checking for program invariant based on their semantics. However, in their semantics, each transition adds one operation instead of one transaction. This significantly increases the searching space. Kaki et al. [2017] proposed a state-based semantics of which the abstract state comprises a global state and client local states. Their semantics allows fine-grained interleaving between clients. They can model SQL isolation levels and `SI` but not models weaker than `SI`. Crooks et al. [2017] proposed a trace-based operational semantics, a variant of state-based semantics. In their semantics, any new transaction must be *compatible* with respect to the whole trace, Their notion of compatibility is subject to the consistency model definition. All those general operational semantics only focused on one side of the challenges either program analysis or verification of implementations: Nagar and Jagannathan [2018] and Kaki et al. [2017] focused on program analysis, and Crooks et al. [2017] focused on verification of implementations.

We propose a new state-based operational semantics that models the interface of the distributed systems, with the focus on the client-observable behaviours. Our abstract state comprises a *global*, *centralised* multi-versioning key-value store (kv-store) which an abstraction of the state of a distributed system, and several *partial* client views which contain partial information over this kv-store. We have a standard interleaving semantics in that a client commits an atomic transaction in each transition. Every transition is constrained by an *execution test* that determines if the client with a given view is allowed to commit the transaction. The execution test gives rise to a consistency model. We specify many well-known consistency models using execution tests and validate them with respect to the well-known axiomatic definitions on abstract executions. We believe that our semantics is an ideal mid-point for verifying implementations and reasoning about client programs, in contrast to other operational semantics that focus on one of the tasks. We prove two implementation protocols the COPS which is a fully replicated database implementing causal consistency, and the Clock-SI which is a partitioned database implementing snapshot isolation, using trace refinement. We show that our abstract states are a faithful abstraction of the machine states of implementation protocols. We then prove invariants of several client programs. For example, we are able to prove the robustness of a single counter library against `PSI` and a multi-counter and a banking library against `SI`. We also prove the correctness of a lock paradigm under `PSI`, despite not robust.

1.1 Contributions

paragraphOperational Semantics. The main contribution of this thesis is *an interleaving operational semantics* that focuses on the interface and client-observable behaviours of distributed systems. The state of this semantics consists of a *global* centralised multi-versioning key-value store (kv-store), which is the abstraction of the machine states of databases, and client-local *partial* views on the kv-store, which models the information that clients observe. For each

transition, a client commits a transaction *atomically*. Our semantics is parametrised by an *execution test*, which gives rise to a consistency model. An execution test determines if a client with a given view is allowed to commit a transaction. We are able to define many weak consistency models employed by distributed databases that satisfy the *snapshot property*[1], including *causal consistency*, *snapshot isolation*, *parallel snapshot isolation* and *serialisability*. We are not able to model few well-known isolation levels such as read committed. However, we focus on protocols and applications employed by distributed databases, most of which guarantee *snapshot property*. Given our semantics, we identify a new consistency model that lies between (stronger than) parallel snapshot isolation (`PSI`) and (weaker than) snapshot isolation (`SI`). We name it as *weak snapshot isolation*. This model disallows *long fork* anomaly, which is the litmus test distinguishing `PSI` and `SI`.

Validation. We validate our operational semantics using abstract executions in the sense that: our definitions of consistency models using execution tests are proven to be correct with respect to the axiomatic definitions on abstract executions; the set of kv-stores induced by an execution test for a consistency model is equivalent to the set of abstract executions induced by the axiom for the consistency model. A kv-store is equivalent to an abstract execution if it contains the same set of transactions. As a technical contribution, we provide an alternative operational semantics on abstract executions, which can be parametrised by a set of axioms that gives rise to a consistency model, where each transition in the abstract execution operational semantics is committed in an atomic step. Our goal is to prove, given a consistency model, the set of the reachable kv-stores are equivalent to the set of the reachable abstract execution, which can be derived by equivalence between traces in kv-store and in abstract execution. Instead of directly working on traces, we provide a *soundness* constructor that constructs a trace in our kv-store operational semantics to a trace on abstract executions, and a *completeness* constructor that is for the opposite direction. With the help of these constructors, we can lift soundness and completeness conditions to the level of traces, and therefore, to the level of the set of kv-stores and the set of abstract executions.

Applications. We showcase two applications of our operational semantics: the verification of implementations; and the invariant properties of client programs. Our abstract machine state, a multi-versioning kv-store and client views, is a faithful abstraction of the machine state of implementation protocols. We verify COPS protocol [Lloyd et al., 2011], a replicated database that implements causal consistency (Section 6.1), by: (1) giving a reference semantics of the COPS protocol; (2) normalising traces induced by the COPS semantics in which transactions are executed atomically; (3) encoding the normalised traces to our kv-store traces; and (4) proving every transition in the kv-store traces satisfies the execution test for causal consistency.

[1]The snapshot property is also known as atomic visibility in that a transaction takes a snapshot at the beginning and commits the effect at the end.

Similarly, we verify Clock-SI [Du et al., 2013], a partitioned database that implements snapshot isolation (Section 6.2). Our operational semantics tracks client-observable history. This enables us to prove invariant properties of client programs (Chapter 7), for example, robustness. Most existing techniques for robustness are based on declarative semantics, which requires the final results obtained by executing client programs to the end. We prove robustness by proving an invariant of each transition. We apply this technique to prove the robustness of a single counter against parallel snapshot isolation (`PSI`) and multiple counters and banking examples against snapshot isolation (`SI`). We find general conditions that guarantee robustness against weak snapshot isolation and therefore any stronger models such as `SI`. To our knowledge, our robustness results are the first to take into account client sessions. With sessions, we demonstrate that multiple counters are not robust against `PSI`. Interestingly, without sessions order, it can be shown that multiple counters are robust against `PSI` (model weaker than `SI`) using static-analysis techniques [Bernardi and Gotsman, 2016]. These techniques are known not to be applicable to sessions. Our semantics can prove specific invariant properties of client programs. We show that a lock paradigm, despite being not robust, is correct under update atomic, a consistency model disallowing conflict writes.

1.2 Thesis Outline

Chapter 2 Background. We start with the background. We briefly summary several well-known consistency models defined by reference implementations [Berenson et al., 1995; Du et al., 2013; Sovran et al., 2011; Raad et al., 2018; Hutto and Ahamad, 1990; Petersen et al., 1997; Lloyd et al., 2011; Liu et al., 2018; Terry et al., 1994; Bailis et al., 2014], such as *causal consistency*, *parallel snapshot isolation* and *snapshot isolation*. However, these definitions using implementations are restricted in specific settings. Researchers then proposed several formal semantics, including declarative and operational approaches. We cover two well-known declarative semantics, dependency graphs [Adya, 1999] and abstract executions [Burckhardt et al., 2012; Cerone et al., 2015a]. Many weak consistency models can be defined on these two declarative models and they are proven to be the same [Cerone et al., 2017]. We only find few general operational semantics including graph-based [Nagar and Jagannathan, 2018; Doherty et al., 2019], state-based [Crooks et al., 2017; Kaki et al., 2017] and log-based [Koskinen et al., 2010; Koskinen and Parkinson, 2015] operational semantics. All those general semantics are useful to verify implementation protocols or reason about client programs. However, none of them, we believe, are able to tackle both challenges.

Chapter 3 Overview. We give: (1) an overview of our operational semantics; (2) definitions of various consistency models in our semantics; (3) validation of them with respect to well--known definitions on abstract executions; and (4) two applications of our operational semantics, verifying implementations and proving invariant properties of client programs. First, we de-

scribe the three key concepts of our operational semantics, *kv-stores*, *views* and *execution tests* using two examples, the single and multiple counters. We show that how to present the state of these counters using our *global* kv-store, and how to commit a transaction to the kv-store with a *partial* view. We describe how to define several weak consistency models via the execution tests and how these execute tests restrict the execution of the single and multiple counters. We then summarise the two applications of our semantics: that is, we verify the COPS protocol, a replicated database that implements causal consistency; and we prove robustness results of the single, multiple counters, and a banking application via proving invariant properties, and prove the correctness of a lock paradigm, despite being not robust.

Chapter 4 Operational Semantics. We formally define *centralised kv-store*, *partial views*, our *interleaving operational semantics* and *execution tests*. Our kv-store abstracts the machine states of distributed databases. Views abstract the information that is observable to clients. Our semantics describes the interface, especially the client-observable behaviour, of distributed key-value stores. Our semantics is parametrised by an *execution test*, which determines if a client is allowed to commit a transaction with a given view. We then define many well-known consistency models using our execute tests. All these models satisfy *snapshot property* such as causal consistency (`CC`), parallel snapshot isolation (`PSI`), snapshot isolation (`SI`) and serialisability (`SER`). We also identify a new consistency model that sits between `PSI` and `SI` and we think that it retains good properties of both. We call this new consistency model *weak snapshot isolation* (`WSI`).

Chapter 5 Correctness of Consistency Models. We prove definitions of consistency models using our execution tests are correct with respect to the well-known declarative semantics on abstract executions. We propose an alternative operational semantics on abstract executions. This alternative semantics is parametrised by an axiom, which gives rise to a consistency model. We show that, given an axiom $\mathcal{A}$, the set of abstract executions induced by the alternative semantics is the same set induced by directly applying the axiom on abstract executions. We then show the set of kv-stores induced by our semantics under an execution test `ET` is equivalent to the set of abstract execution tests the alternative semantics under an axiom $\mathcal{A}$. A kv-store is equivalent to an abstract execution if they contain the same transactions. This equivalence is defined by: (1) showing a bijection between our kv-stores and dependency graphs; and (2) using the result from [Cerone et al., 2017], which proved the connection between dependency graphs and abstract executions. For technical contribution, we propose soundness and completeness constructors in the sense that, if `ET` and $\mathcal{A}$ satisfy the soundness and completeness conditions, these constructors lift these conditions to the level of traces. We use these two constructors prove the correctness of our definitions of consistency models.

Chapter 6 Applications: Verification of Implementations. In this and the next chapter, we explore two immediate applications of our operational semantics: the correctness of specific implementation protocols and the establishment of invariant properties such as robustness of client applications. By contrast, in the literature these tasks tend to be carried out in *different* declarative formalisms: clients are analysed using dependency graphs [Fekete et al., 2005; Cerone and Gotsman, 2016; Bernardi and Gotsman, 2016; Cerone et al., 2015b; Nagar and Jagannathan, 2018]; implementation protocols are verified using abstract execution graphs [Burckhardt et al., 2014; Cerone et al., 2015a].

In this chapter, we verify two implementation protocols against our definitions of consistency models: the COPS protocol which is fully replicated key-value stores [Lloyd et al., 2011] satisfying causal consistency; and the Clock-SI protocol which is partitioned key-value stores [Du et al., 2013] satisfying snapshot isolation. We demonstrate that the traces of these implementation protocols can be refined to traces in our semantics, and show that each transition in these traces satisfies the execution test.

Chapter 7 Applications: Invariant Properties of Client Programs. We prove the robustness of the single counter against parallel snapshot isolation. We then show the robustness of the multi-counter and a banking applications Alomari et al. [2008] against snapshot isolation. These applications satisfy a general condition which guarantees the robustness of weak consistency model and hence any stronger models such as snapshot isolation.

Chapter 8 Conclusion. To conclude, we summarise this thesis and the applications and discuss the future work.

Chapter 2

Background

Distributed systems must make a trade-off between high partition-tolerance, high availability and strong consistency, This is due to the CAP theorem [Gilbert and Lynch, 2002], which states that systems can at most achieve two out of the three properties. Most systems choose partition-tolerance and availability, and drop strong consistency. Those systems are thus implementing weak consistency models. In fact, many well-known consistency models were originally invented by implementers in that the actual implementations of these consistency models are simple and have good performance in practice (Section 2.1).

However, it is impossible to compare different consistency models that are defined by providing implementation protocols. Much work has been done to formalise consistency models, both declaratively and operationally. Many formalisms focus on consistency models that were designed for centralised or distributed databases. More recently, these formalisms have been adapted to model weak memory, which can be seen as a type of consistency model. In this thesis, we focus on the consistency models that are mainly used in databases. On the declarative side (in Sections 2.2 and 2.3), the two main *general* formalisms are dependency graphs [Adya, 1999] (Section 2.2) and abstract executions [Burckhardt et al., 2012; Cerone et al., 2015a] (Section 2.3). Both of them provide a unified semantics for formulating different consistency models. [Cerone et al., 2017, 2015a] showed that the definitions of several consistency models on these two formalisms are equivalent. We will informally explain these two formalisms in this chapter and give the formal definitions in Chapter 5. Both declarative formalisms are important to us: (1) the machine states of our semantics are inspired by dependency graphs; and (2) our definitions of consistency models are inspired by the definitions on dependency graphs and proven to be sound and complete with respect to the definitions on abstract executions.

On the operational side (in Section 2.4), There has been little work on *general* operational semantics for describing a range of consistency models (Section 2.4). There are graph-based [Nagar and Jagannathan, 2018], state-based [Kaki et al., 2017; Crooks et al., 2017] and log-based [Koskinen et al., 2010; Koskinen and Parkinson, 2015] operational semantics in the literature. Nagar and Jagannathan [2018] proposed an operational semantics on abstract executions where

each transition appends a single read or write operation to abstract executions. They capture consistency models by ruling out the invalid results with respect to the declarative definitions on abstract executions. They developed a model-checking tool that can check robustness. However, we believe that it is difficult to reason specific invariants of client programs, because it is unknown how to efficiently encode these invariants in abstract executions. Kaki et al. [2017] proposed an operational semantics with explicitly global and local states. However, these semantics are designed to SQL isolation levels, but we think they cannot capture some models in distributed systems such as parallel snapshot isolation. In their models, because clients must agree on the global state, despite they might not observe all the state yet. Crooks et al. [2017] proposed an operational semantics that can capture models from snapshot isolation to serialisability. Each transition in their model commits a new transaction t, if t is compatible to the entire trace in the sense that the any read operation in t must read from previous transactions in the traces, and t is allowed to read from those transactions. This check gives rise to a consistency model. Their models are useful to verifying implementation because of the presence of the total order over all transactions. However, it is difficult to use their model to prove invariants of client programs, which requires client observable history instead of the entire history. Two log-based operational semantics [Koskinen et al., 2010; Koskinen and Parkinson, 2015], similar to our operational semantics, have a global state and client local states. These two models capture the interface between clients and implementations. However, they focus on serialisability and we believe it is difficult to generalise their models to other consistency models.

In contrast to other operational semantics, we believe that our semantics is the first state-based operational semantics that are suitable for verifying implementations of consistency models and reasoning about client programs. Our abstract states are multi-versioning stores that contain client-observable history, which enables us to prove invariants of client programs. Our multi-versioning stores with meta-data are a faithful abstraction of many machine states of implementations. This allows us to verify implementation protocols. To validate our semantics, we propose an alternative graph-based operational semantics that is similar to some graph-based semantics [Nagar and Jagannathan, 2018] in the literature. This alternative graph-based semantics helps us build the correspondence between our state-based operational semantics and the declarative semantics.

2.1 Consistency Models

Many weak consistency models were invented by database implementers to tackle real-world constraints and maximise performance. The semantics of a *specific consistency model* can be captured by a reference implementation for a specific setting, or described by forbidden behaviours, known as *anomalies*.

Serialisability (SER). This model is de facto the standard consistency model for centralised databases in the sense that many real-world transactional applications are designed with respect to serialisability. It requires that all committed transactions can be seen as appear one after another, despite that they may execute in a fine-grained interleaving way. Serialisability is the strongest consistency model if one does not consider aborted transactions[1]. This model was originally defined by implementation strategies using locks [Papadimitriou, 1986; Eswaran et al., 1976]. Researchers proposed then the concept of history, a trace of fine-grained read and write operations, and gave a formal definition of serialisability [Papadimitriou, 1986; Eswaran et al., 1976]: there exists an equivalent history where transactions appear one after another.

SQL isolation levels. The four well-known SQL isolation levels (ANSI SQL-92), *read uncommitted*, *read committed*, *repeatable read* and *serialisability*, were summarised in [Berenson et al., 1995]. In a centralised database where transactions are executed in a fine-grained way:

(1) *read uncommitted* allows transactions to read uncommitted values;

(2) *read committed* requires transactions read committed values, but two reads to the same object in the same transaction may be different; and

(3) *repeatable read* requires transactions consistently read the same committed value.

The ANSI SQL-92 defines these consistency models via anomalies. In contrast to a specific lock-based characterisation of these four isolation levels [Gray et al., 1988], definition by anomalies was intended to allow non-lock-based implementations. However, Berenson et al. [1995] argued that these definitions are ambiguous. To tackle the ambiguity, [Adya, 1999] proposed dependency graphs and gave formal definitions of these four consistency models. These four consistency models are still the standard for many centralised databases. However, these definitions still have strong connections to lock-based implementations. In contrast, in distributed databases where different sites work on different versions, precise implementation of these four consistency models might lead to a huge performance penalty: (1) read uncommitted, read committed and repeatable read allow some level of fine-grained interleaving between transactions executed in different distributed sites, which means more synchronisation messages between sites; (2) serialisability requires constant synchronisation between distributed sites for consensus, which means fewer transactions can run in parallel; (3) a distributed database expects some level of partition intolerance, that is, at least a part of the database can still serve clients when network partition happens. This leads to a discussion about multi-versioning implementation strategy in distributed databases.

[1] A stronger model, opacity, requires that non-committed transactions cannot access inconsistent states [Guerraoui and Kapalka, 2008].

Snapshot Isolation (SI). The discussion of SQL isolation levels leads to another consistency model called *snapshot isolation*. In contrast to lock-based implementations, snapshot isolation used the multi-versioning method from [Bernstein et al., 1986]. Berenson et al. [1995] specified snapshot isolation in a centralised database with an English description of an implementation strategy using transaction snapshot time and commit time. A transaction takes a *snapshot* of the database at its starting time. For read operation, in contrast to reading directly from the database, the transaction reads from the snapshot. When committing, the transaction ensures that there is no conflicting write since the snapshot time, drops the effect of any intermediate steps and only commits the final update of this transaction. Since then, many centralised databases provide snapshot isolation alongside SQL isolation levels. Because snapshot isolation has a strong connection to multi-versioning implementations of databases in the sense that transaction works on their own snapshots of the database, this model fits in distributed databases. Many works used the ideas of snapshot and commit time, and implemented snapshot isolation in various distributed databases, including replicated [Daudjee and Salem, 2006; Elnikety et al., 2005] and partitioned [Du et al., 2013] setting.

Parallel Snapshot Isolation (PSI). It is a natural fit to implement SI in distributed systems in the sense that SI is designed for a multi-versioning implementation strategy. However, SI still requires that two concurrent transactions *observe* the updates of other transactions in the *same* order. This means that transactions on different sites must agree on their relative order, which can be a time-consuming task. To reduce the need of constant synchronisation, Sovran et al. [2011] proposed *parallel snapshot isolation* (PSI), a model slightly weaker than SI. They proposed an implementation strategy for PSI that has similar patterns to those implementations for SI, separating a transaction to three stages: snapshot, internal execution and commit. However, a transaction takes a snapshot of the site instead of the entire database, and the transaction is committed *locally* and then non-deterministically propagated to other replicas. Using this strategy, transactions in different sites may observe updates in a different order, because of the synchronisation delay. Ardekani et al. [2013] gave a formal description of PSI by putting constraints on the read and write operations of each transaction. They also proposed a better implementation on a multi-versioning replicated database where each version maintains explicit dependency sets containing meta-data.

Causal Consistency (CC). Hutto and Ahamad [1990], inspired by the Lamport time [Lamport, 1978] and vector clock [Fidge, 1988; Mattern, 1989], first introduced the notion of causal relation over single read and single write operations on a shared memory model. Both Lamport time and vector clock are implementation techniques used to determine the order between operations. This model requires that if a transaction observes the effect of another transaction t, it must observe all transactions t depends on. Bayou [Petersen et al., 1997] proposed a replicated distributed system that implements CC via a propagation protocol that guarantees that if a

version ν, which is initially accepted by a replica r, has been delivered to a replica r', then all versions happens before ν in the replica r must be delivered to r' before ν. However, Bayou did not provide the full detail but only the focus on the propagation protocol. COPS [Lloyd et al., 2011] refined the idea from [Petersen et al., 1997], implements CC on a multi-versioning replicated database. Besides that COPS provides a fixed interface, that is, single-write and multiple-read transactions, the implementation protocol is very simple with the help of some meta-data. COPS has explicit client state, which is used to maintain the session order. Separately, Belaramani et al. [2006] proposed a specialised propagation protocol for CC with the focus on file systems.

In contrast to SI and PSI, we did not find any partitioned database that implements CC. We believe this is because there is no performance benefit to implement CC in a partitioned database. Instead of focusing on the snapshot and commit time of a transaction, many implementation protocols of CC focus on how to propagate the effect of a transaction from its initial replicas to other replicas.

Read Atomic (RA). Bailis et al. [2014] gave a collection of implementations for *read atomic* (RA) in partitioned databases, using two-phase commit for any write to ensure that writes within the same transaction be propagated to the appropriate partitions. Read atomic ensures that a transaction consistently reads from the same snapshot taken at the beginning and only commits its change at the end. This is also known as *snapshot property* or *atomic visibility*. Many previous consistency models including SER, SI, PSI and CC satisfy RA.

Other models. Cerone et al. [2015a] gave a formal definition of SI using abstract executions, a formal declarative model that we will explain later. The definition splits SI into two parts: *consistency prefix* (CP) and *update atomic* (UA). Consistency prefix requires that if a transaction sees the effect of another transaction t, it must see all transactions committed before t. Note that, the order which transactions commit in CP depends on the resolution policy. Update atomic requires no concurrent write to the same object. Liu et al. [2018] proposed a implementation for update atomic UA in partitioned databases, using two-phase commit and time-stamp.

Terry et al. [1994] introduced four session guarantees, *monotonic read* (MR), *read your write* (RYW), *monotonic write* (MW) and *write follow read* (WFR), defining them via a loose description of implementation strategies on replicated databases. However, we have not found any actual implementation of these models. This is because in practice, we believe, applications on these models have very weak behaviours. On the other hand, many well-known consistency models imply these four session guarantees.

Many consistency models are originally invented using reference implementations on their specific setting. However, implementation details are irrelevant to clients, because they only work

with the interfaces. This means that clients should not need to know the implementation. Another problem of these definitions using reference implementations is that it is difficult to compare between different consistency models, and even between two implementations of the same model. To solve these problems, researchers proposed [Adya, 1999; Burckhardt et al., 2012] many semantics that focus on formal definitions of consistency models.

2.2 Declarative Semantics: Dependency Graphs

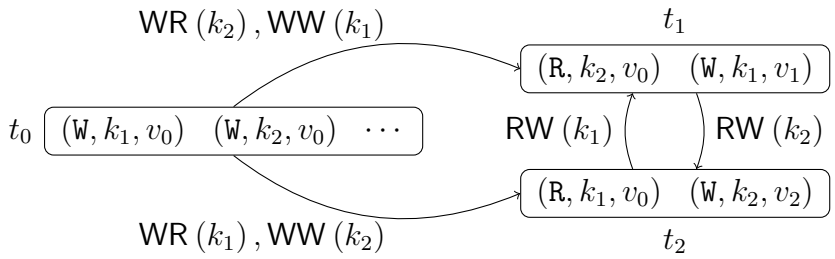
Adya [1999]'s work on *dependency graphs* provided the first general formal model that uniformly specified several consistency models. Previously, many models were defined using reference implementations or *anomalies*[2] [Berenson et al., 1995]. In contrast, with dependency graphs, consistency models were defined by disallowing certain cycles.

Adya [1999] used dependency graphs to give formal definitions of SQL isolation levels and snapshot isolation. These formal, implementation-independent definitions of consistency models provide uniform definitions of SQL isolation levels and snapshot isolation. It becomes possible to compare different consistency models and even explore the gap: for example, Adya [1999] proposed two new consistency models, PL-2+, a consistency model similar to causal consistency, and PL-2L, a variant of read committed.

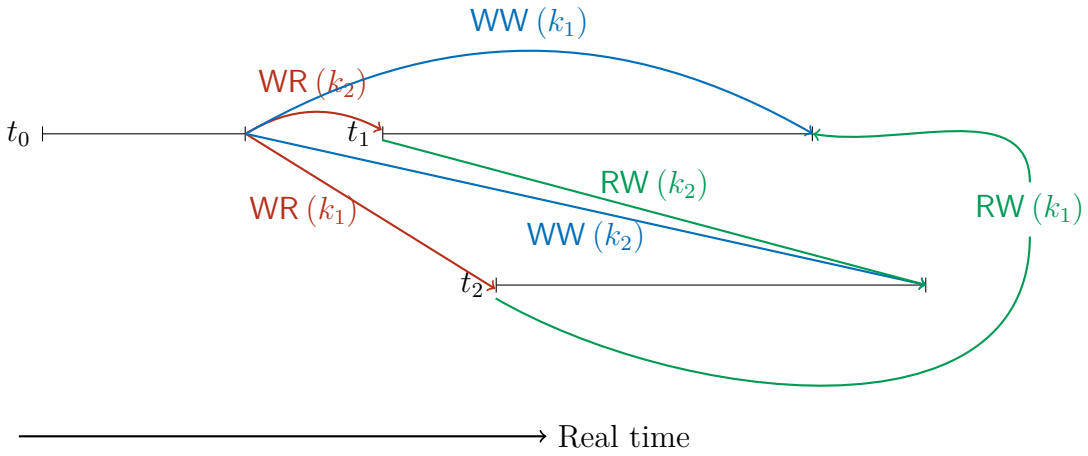
There has been little work that uses dependency graphs to define the consistency models used in distributed systems. Cerone et al. [2015b] explored how to use dependency graphs to specify weak consistency models used in distributed systems, such as parallel snapshot isolation `PSI`. The definitions of consistency models in distributed databases using dependency graphs are difficult to understand. Models like `SI` and `PSI` are originally defined using the concept of snapshot and commit times. There is no direct information about these times in dependency graphs. Instead, database researchers seem to prefer to use abstract executions (we explain in Section 2.3). However, dependency graphs have become a standard tool for studying weak memory, in which accesses to the memory are not serialisible. [Alglave, 2010] adapted dependency graphs to specify weak memory behaviours. Since then, much work [Batty et al., 2011; Doherty et al., 2019; Batty et al., 2016; Lahav et al., 2016, 2017; Batty, 2014] focused on specifying and fixing the C11 memory model.

A dependency graph is a directed graph where each node represents a transaction, consisting of a unique transaction identifier and a set of read and write operations, and edges are labelled with *write-read* (WR), *write-write* (WW) and *read-write* (RW) dependencies. Fig. 2.1a gives an example of a dependency graph. In Fig. 2.1a, there are three transactions, t_0, t_1 and t_2 respectively. The special initialisation transaction t_0 initialise all the keys k_i with the initial value v_0. The *write-read* dependency (WR) determines the source of every read operation. For example, t_1 reads key k_2 with value v_0 written by t_0 in Fig. 2.1a. The *write-write* dependency

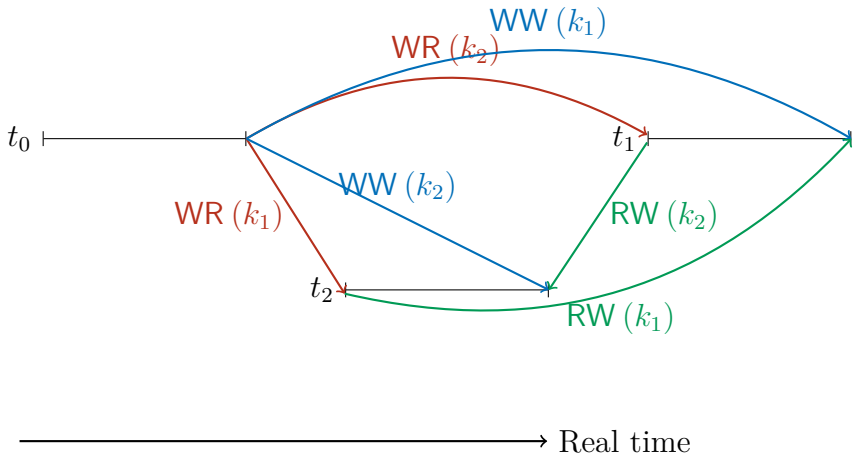
[2] *Anomaly* refers to disallowed behaviour.



(a) An example of a dependency graph, containing the initialisation transaction t_0 and two transactions t_1, t_2.



(b) An example time line in a centralised database for Fig. 2.1a (WR in red, WW in blue and RW in green).



(c) An example time line in a distributed database for Fig. 2.1a, where t_1 and t_2 are committed to different sites (WR in red, WW in blue and RW in green).

Figure 2.1: An example of a dependency graph and time lines

(WW) is a total order per key k, over all transactions that wrote to the key k. For example, $(t_0, t_1) \in \mathsf{WW}$ on the key k_1. The *read-write* anti-dependency (RW) is derived from other two dependencies, in that if $(t_0, t_2) \in \mathsf{WR}$ and $(t_0, t_1) \in \mathsf{WW}$ on a key k, then $(t_2, t_1) \in \mathsf{RW}$. Intuitively, if $(t_2, t_1) \in \mathsf{RW}$, then t_2 read a value that has been overwritten by t_1. The read-write anti-dependency is crucial for specifying certain weak consistency models such as snapshot isolation.

Fig. 2.1b presents the intuition of the dependency relations in a centralised database where all transactions agree on a global clock. This global clock is often the physical time in the centralised database. The write-read relation, $(t_0, t_1) \in \mathsf{WR}$, intuitively means that the commit of t_0 happens before the start of t_1 in a centralised database in Fig. 2.1b. The write-write relation, $(t_0, t_1) \in \mathsf{WW}$, means that the commit of t_0 happens before the commit of t_1. However, this relation does not provide any information about the start points of transactions. The write-write relation in a centralised database is usually determined by the physical as shown in Fig. 2.1b. Last, the read-write relation, $(t_2, t_1) \in \mathsf{RW}$, means that the starts of t_2 happens before the commit of t_1, thus t_2 does not observe the update of t_1.

In centralised databases, these dependency relations can be determined by the start and commit points of transactions. However, there is, in general, no total order over the start and commit points of transactions in distributed databases. In these distributed databases, the write-read dependency is directly determined by the observation of transactions in the sense that if a transaction reads from another transaction, then there is a WR edge between them. For example, in Fig. 2.1c, t_1 reads the initialisation transaction but not t_2. The write-write dependency is determined by the *resolution policy*, which determines the order between two transactions, if these transactions write to the same key. Note that an order determined by a resolution policy may not agree with the actual physical time. For example, in a distributed database, assume two transactions, t_1 and t_2, commit to two distinct sites. The resolution policy may decide t_1 abstractly commits before t_2, even thought t_2 commits after t_1 in the actual physical time. One widely used resolution policy is *last-write-wins* [Vogels, 2009]. This policy states that if two transactions write to the same key, the transaction with a greater abstract time should be ordered after the transaction with the smaller abstract time. For example, in Fig. 2.1c, there are WW edges from t_0 to t_1 and t_2 respectively. In a distributed database, this policy is implemented using Lamport time or vector time, which provides a partial order over transactions. This partial order can be extended to a total order over all transactions.

Dependency graphs give uniform definitions for different consistency models where each consistency model is defined by disallowing certain cycles. For example, *serialisability* (`SER`) enforces that dependency graphs contain no cycle. This means that there exists a total order among all transactions by extending the existing relations to a total order. *Snapshot isolation* (`SI`), a model that is weaker than `SER`, only allows cycles with two adjacent RW edges. Adya [1999] proved that this acyclic condition corresponds to the English description in [Berenson et al., 1995]; however, the intuition is not apparent. The dependency graph in Fig. 2.1a is allowed by `SI` but not by `SER`. With dependency graphs, the definitions of *parallel snapshot isolation* and *causal consistency* models, two models that are mainly used in distributed systems, are not straightforward. We give the definitions in Section 5.5.

Dependency graphs are suitable for reasoning about client programs. This is because the three dependency relations in dependency graphs can be statically over-approximated. Assume the

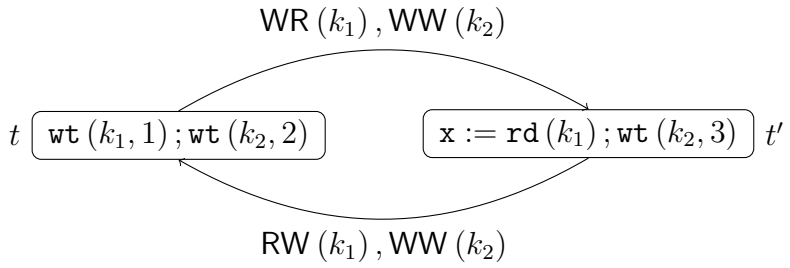


Figure 2.2: An example of a static dependency graph

following program:

$$\mathtt{P}_{\mathtt{static}} \stackrel{def}{=} t : \left[\mathtt{wt}\,(k_1, 1)\,;\mathtt{wt}\,(k_2, 2)\right] \| t' : \left[\mathtt{x} := \mathtt{rd}\,(k_1)\,;\mathtt{wt}\,(k_2, 3)\right],$$

where the commands $\mathtt{x} := \mathtt{rd}\,(k_1)$ and $\mathtt{wt}\,(k_2, 3)$ means read from key k_1 and write to key k_2, and the square bracket means a transaction. For brevity, in the explanation, we annotate transactions with transaction identifiers, such as t, t'. In the program, there is a static write-read edge and a static read-write edge, $(t, t') \in \mathsf{WR}$ and $(t', t) \in \mathsf{RW}$, on k_1, and static write-write edges $(t, t'), (t', t) \in \mathsf{WW}$ on k_2, depicted in Fig. 2.2. These static relations contain all the possible dependency relations of these two transactions. This means that, if certain cycles do not exist in the static graph, it must be absent in the actual dependency graphs. Many robustness[3] and transaction chopping[4] techniques have been described as properties of these three static relations [Fekete et al., 2005; Bernardi and Gotsman, 2016; Cerone and Gotsman, 2016].

Dependency graphs are useful for reasoning about client programs. However, it appears to be difficult to verify implementation protocols using the consistency definitions on dependency graphs. Our state-based operational semantics, in contrast, is suitable for verifying implementation protocols. We believe that our machine states provide a faithful abstraction to the machine states of many protocols, and we can verify transition in these protocols step by step.

2.3 Declarative Semantics: Abstract Executions

The baseline model for many distributed systems is eventually consistency [Vogels, 2009], which states that if there are no new updates, then eventually all accesses return consistency values, for example, the most up-to-date values. It is difficult to capture that eventually all transaction observe the up-to-data value using dependency graphs. Burckhardt et al. [2012], therefore, introduced *abstract executions*. To remedy this, Burckhardt et al. [2012] first defined a *history*

[3]Robustness: transactions executing under a weak consistency model, have the same behaviours as if executing under serialisability.

[4]transaction chopping: chopping a transaction into several smaller transactions does not introduce new behaviours.

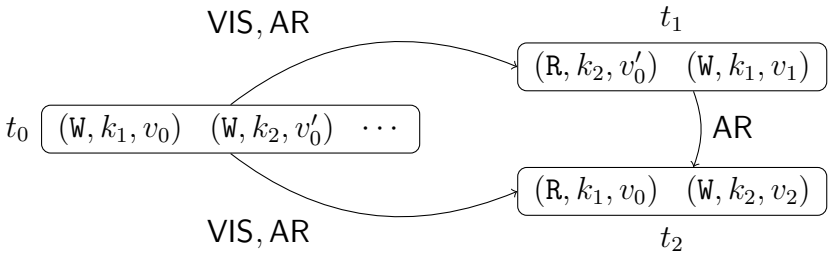


Figure 2.3: An example of an abstract execution

that contains all single-read and single-write operations of all transactions and a partial order over all single operations. This history is similar to that in [Papadimitriou, 1986], which was used to defined serialisability. If the partial order on the history can be abstracted to two relations, a *visibility relation* which determines for each transaction the set of observable transactions, and an *arbitration relation* which is a total order in which transactions take effect, then the history satisfies eventual consistency. The visibility relation allows a transaction to read old values. The arbitration relation determines the final state of the overall distributed system: that is, the state that transactions *eventually* should observe. The abstract executions in [Burckhardt et al., 2012] are defined in the level of single operations in transactions to distinguish serialisability and eventual consistency. Burckhardt [2014] adapted abstract executions to *replicated data types*. In a replicated data type, instead of arbitrary transactions, the primitive operations are a set of high-level APIs, for example, push and pop. They defined well-known weak consistency models, such as causal consistency, based on replicated data types. Separately, Cerone et al. [2015a] adapted abstract executions by only working at the level of atomic transactions. With this approach, they were able to define weak consistency models that satisfy snapshot property such as snapshot isolation and causal consistency. All these definitions from [Burckhardt, 2014; Cerone et al., 2015a] restrict the visibility relations. Since we focus on transactions and consistency models with snapshot property, we will work with the formal definition of abstract executions from [Cerone et al., 2015a].

Burckhardt et al. [2012] used abstract executions, a graph-based model that has different types of edges from dependency graphs, to distinguish eventual consistency from serialisibility. Later on, more consistency models, which lie between eventual consistency and serialisability, were defined in this framework, such as session-oriented consistency models (on replicated data types [Burckhardt, 2014; Burckhardt et al., 2014]), causal consistency (on replicated data types [Burckhardt, 2014; Burckhardt et al., 2014] and transactions [Cerone et al., 2015a]), parallel snapshot isolation (on transactions [Cerone and Gotsman, 2016]) and snapshot isolation (on transactions [Cerone et al., 2015a]).

An abstract execution [Cerone et al., 2015a] is a directed graph with nodes that represent transactions and edges labelled with *arbitration* and *visibility* relations. The *arbitration* relation, also known as *arbitration order*, is a total order over all transactions, which determines the order

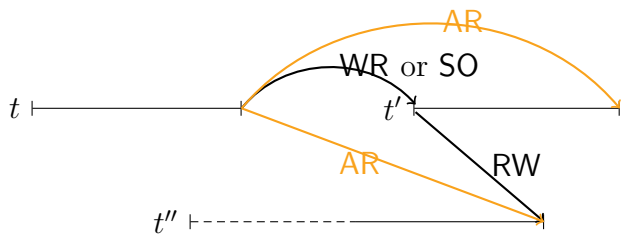
in which transactions take effect. For example, in Fig. 2.3, there is arbitration order $t_0 \xrightarrow{\mathsf{AR}} t_1 \xrightarrow{\mathsf{AR}} t_2$. This arbitration order intuitively corresponds to the *commit order* in centralised databases. In distributed databases, the commit order is a partial order since transactions might commit to different sites. If there is no commit order between two transactions, then they might have conflicting writes. However, under eventual consistency, conflict transactions must be ordered by resolution policy, which gives rise of total order among all transactions, and henceforth the arbitration order. For example, in Fig. 2.3, transactions t_1 and t_2 may be committed to different sites and the arbitration order from t_1 to t_2 is determined by a certain resolution policy. The *visibility* relation determines, for every transaction, the set of visible transactions, which then determines the actual read value for each key. For example, in Fig. 2.3, both t_1 and t_2 only see the initialisation transaction t_0 and both of them read the initial value v_0 for keys k_1 and k_2 respectively. Note that, for WR in dependency graphs, if $(t_0, t_1) \in \mathsf{WR}$, it means t_1 *must* read the effect of t_0 in the sense that t_1 have actually read a value written by t_0. In contrast, if $(t_0, t_1) \in \mathsf{VIS}$, it means that t_1 is *aware* the effect of t_0 and *may* read from t_0. Hence, researchers normally use terms like t_1 '*observes*' or '*sees*' t_0 to capture the awareness but not necessarily direct read. When the set of visibility transactions contains several transactions that write to the same key, the actual read value depends on the resolution policy. One common resolution policy is *last-write-wins* stating that the latest write for a key overwrites any previous write for the same key. This means, a transaction always reads from the latest visible transaction.

In abstract executions, consistency models are defined by putting constraints on the visibility edges. In contrast to dependency graphs, the definitions of many consistency models using abstract executions, especially those models that mainly used in distributed databases, are more intuitive and thus have become the de facto standard definitions. This is because AR is a total order. Cerone and Gotsman [2016]; Cerone et al. [2017] proposed a technique that converts the definitions of consistency models using abstract executions to the equivalent definitions using dependency graphs, and vice versa. For example, *serialisability* (SER) is defined by $\mathsf{VIS} = \mathsf{AR}$ in abstract executions [Gilbert and Lynch, 2002], and the relation[5] $(\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW} \cup \mathsf{SO})^+$ contains no cycle in dependency graphs [Adya, 1999]. Another example is *consistency prefix* (CP), which states that if a transaction t sees some transaction t', then t must see all transactions committed before t'. This can be captured by $\mathsf{AR}; \mathsf{VIS} \subseteq \mathsf{VIS}$[6] in abstract executions [Cerone et al., 2015a]. The definition of CP on dependency graphs requires the relation $\big((\mathsf{SO} \cup \mathsf{WR}); \mathsf{RW}^? \cup \mathsf{WW}\big)^+$ be acyclic [Cerone et al., 2015a]; this relation approximate the relation AR in abstract executions as shown in Fig. 2.4. Hence, if $\big((\mathsf{SO} \cup \mathsf{WR}); \mathsf{RW}^? \cup \mathsf{WW}\big)^+$ is acyclic in dependency graphs, then $\mathsf{AR}; \mathsf{VIS} \subseteq \mathsf{VIS}$ holds in the equivalent abstract executions.

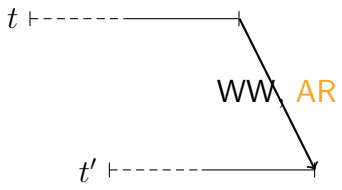
In contrast to dependency graphs, abstract executions are more suitable for verifying implemen-

[5] R^+ denotes the transitive closure of R.

[6] $\mathsf{AR}; \mathsf{VIS} \stackrel{def}{=} \{(t, t') \mid \exists t''. (t, t'') \in \mathsf{AR} \wedge (t'', t') \in \mathsf{VIS}\}$.



(a) Relations WR, SO, WR; RW or WR; RW with respect to AR (AR in orange)



(b) Commit before: WW

Figure 2.4: Example of relation $\left((\mathsf{SO} \cup \mathsf{WR}); \mathsf{RW}^? \cup \mathsf{WW}\right)^+$

tations as the arbitration relation is a faithful abstraction of many implementation mechanisms such as Lamport time [Lamport, 1978] and vector clock [Fidge, 1988; Mattern, 1989]. The Lamport time and vector clock determines a partial order over all transactions, which can then be extended to the arbitration relation. The visibility relations is a faithful abstraction explicit dependency meta-data used in implementations, which often contain information such as the snapshot times of transactions. However, it is difficult to use abstract executions to reason about client programs, since the model requires a total order among all transactions. In contrast, it is better to use dependency graphs which only contain client-observable relations.

2.4 Operational Semantics

Unlike declarative approaches, there has been little work on general operational semantics for describing a range of consistency models, state-based [Kaki et al., 2017; Crooks et al., 2017], graph-based [Nagar and Jagannathan, 2018], and log-based operational semantics [Koskinen et al., 2010; Koskinen and Parkinson, 2015]. Although these log operational semantics can only model serialisability, they are similar to our general operational semantics in the sense that they focus on client observable behaviours.

Kaki et al. [2017] proposed an operational semantics for SQL transaction programs under the consistency models given by the standard ANSI/SQL isolation levels: namely, read uncommitted, read committed, repeatable read and snapshot isolation. Their operational semantics accesses a global, centralised store γ, and each client has its own local state σ, which interacts with the global store in a fine-grained way.

$$\begin{aligned} &\gamma, \sigma \xrightarrow{t:\{(\mathtt{W},k_1,v)\}} \gamma, \sigma\left[k_1 \mapsto v\right] \xrightarrow{\texttt{synchronisation}} \\ &\quad \gamma\left[k_1 \mapsto v\right], \sigma\left[k_1 \mapsto v\right] \xrightarrow{t:\{(\mathtt{W},k_2,v')\}} \gamma\left[k_1 \mapsto v\right], \sigma\left[k_1 \mapsto v\right]\left[k_2 \mapsto v'\right] \end{aligned} \tag{2.1}$$

This means that, when a client executes a transaction, for example t in Eq. (2.1), with respect to its own local state σ, the transaction may synchronise σ with γ before the commit of the transaction. This means other transactions may observe the intermediate steps of t. Their

semantics is used to develop a program logic and prototype tool for reasoning about client programs. They have not compared their definitions of several isolation levels (consistency models) with the well-known definitions given in the literature. They capture models such as snapshot isolation (SI), but not weaker models such as parallel snapshot isolation (PSI) and causal consistency (CC) which are important for distributed databases. In their framework, any change to the global store is immediately made available to all clients. This means that their semantics is not suitable to capture weak consistency models such as PSI or CC where clients may have different views on the system.

Crooks et al. [2017] proposed a trace semantics over a global centralised store, where the behaviour of clients is formalised by the observations they make on the totally-ordered history of states. They introduced concepts called *read states* and *commit tests* respectively. For each read operation $(\mathtt{R}, k, v)$ in a transaction t, the read state is a state s where the k has value v. The set of read states of a transaction t is the *union* of read states for all read operations in t. Read states are similar to client views in our operational semantics, each of which determines the snapshot of a transaction. A commit test determines, given the read states of a transaction t, if t is allowed to commit. This is similar to the execution test in our semantics which determines, given a client view, if a transaction is allowed to commit. An example trace of [Crooks et al., 2017] that produces *write skew* anomaly is of the form:

$$s : \{k_1 \mapsto v_0, k_2 \mapsto v_0\} \xrightarrow{t:\{(\mathtt{W},k_1,v),(\mathtt{R},k_2,v_0)\}} s' : \{k_1 \mapsto v, k_2 \mapsto v_0\} \xrightarrow{t':\{(\mathtt{R},k_1,v_0),(\mathtt{W},k_2,v')\}} s'' : \{k_1 \mapsto v, k_2 \mapsto v'\}. \quad (2.2)$$

The initial state s contains two keys k_1 and k_2, both of which have the initial value v_0. Each transition in the trace corresponds to a transaction. The first transaction t reads the initial value of k_2 and update the value of k_1 to v. Since there is only the initial state, the read state of t for k_2 must be the singleton set $\{s\}$. The second transaction t' reads the initial value of k_1 and updates the value of k_2 to v'. Here the read states of the read operation $(\mathtt{R}, k_1, v_0)$ in transaction t' is the singleton set $\{s\}$, which means that this transaction did not read the latest value for k_1. Given this trace, each transaction must satisfy the *commit test* which gives rise to a consistency model. For example the commit test for SI enforces that, for each transaction t, there must exist a read state s such that:

(1) for any read operation, $(\mathtt{R}, k, v)$, the read value v matches $s(k)$; and

(2) for any write operation, $(\mathtt{W}, k, v)$, the value of the key k has not been modified since the state s.

For example, Eq. (2.2) is a valid SI trace. Note that each read operation has its own set of read states, and a transaction might read from different read states for different keys. This allows them to model consistency models that are weaker than SI. Given the *read states* and

commit tests, they can capture a wide range of consistency models from *read committed* to *serialisability*.

They use their semantics to demonstrate the equivalence of several implementation-specific definitions of `SI`. In their semantics, one-step trace reduction is determined by the whole previous history of the trace and especially the total order of all transactions. However, the usefulness of their approach for analysing client programs is not immediately apparent, since observations made by their clients involve information that is not generally available to real-world clients, such as the total order in which transactions commit.

For graph-based operational semantics, Nagar and Jagannathan [2018] proposed an operational semantics over abstract executions $\mathcal{X}$, rather than concrete centralised stores, in order to build a model-checking tool for proving the robustness of applications against a given consistency model:

$$\mathcal{X} \xrightarrow{t:\{(\mathtt{W},k_1,v)\}} \mathcal{X}' \xrightarrow{t':\{(\mathtt{R},k_1,v')\}} \mathcal{X}'' \xrightarrow{t:\{(\mathtt{W},k_2,v')\}} \mathcal{X}'''. \tag{2.3}$$

Each transition in their semantics is a single operation in a transaction. Their traces may not always be able to progress, because of the presence of the intermediate steps of transactions. They are able to capture weaker consistency models such as `PSI` and `CC`. However, although they focus on consistency models with the snapshot property, a transaction reads from an atomic snapshot of the database and commits atomically, their semantics allows for the fine-grained interleaving of operations in different transactions. We believe that this results in an unnecessarily complicated semantics and it also increases the search space. In contrast, we propose an operational semantics on abstract executions in which each transition is an atomic transaction. This operational semantics is an intermediate step for proving our definitions of consistency models on kv-stores are correct with respect to the declarative definitions on abstract executions. In our abstract execution semantics, the scenario in Eq. (2.3) may look like as the following:

$$\mathcal{X} \xrightarrow{t:\{(\mathtt{W},k_1,v),(\mathtt{W},k_2,v')\}} \mathcal{X}^* \xrightarrow{t':\{(\mathtt{R},k_1,v')\}} \mathcal{X}^{**}.$$

There have been some log-based operational semantics [Koskinen et al., 2010; Koskinen and Parkinson, 2015] that focus on client observable behaviours. Koskinen et al. [2010] introduced two semantics for serialisability, a pessimistic style and an optimistic style. They did not limit themselves to simple read and write operations within transactions, but also allow operations on abstract data types (ADTs). They model the state of a database as a log that contains a list of operations and they achieve serialisability by enforcing some operations be left mover operation (in the pessimistic case) or right mover operation (in the optimistic case). Left mover and right mover operations state that an operation can be moved to the left (earlier) and the right (later) respectively in the log without changing behaviours of other operations.

Koskinen and Parkinson [2015] proposed a model that further unifies several implementations of serialisability. Their semantics consists of a global shared log g and client local logs l, where a log is a list of operations. New operations are appended to the client local log, *pushed* to the shared log and *pulled* from the shared log. An example trace is given in the following:

$$\begin{aligned} g : [o_0, \cdots, (\mathtt{W}, k_2, v')], l : [o'_0, \cdots, o'_m] &\xrightarrow{t:\mathtt{pull}} g : [o_0, \cdots, (\mathtt{W}, k_2, v')], l' : [o'_0, \cdots, o'_m, (\mathtt{W}, k_2, v')] \\ &\xrightarrow{t:\{(\mathtt{W}, k_1, v)\}} g : [o_0, \cdots, (\mathtt{W}, k_2, v')], l'' : [o'_0, \cdots, o'_m, (\mathtt{W}, k_2, v'), (\mathtt{W}, k_1, v)] \\ &\xrightarrow{t:\mathtt{push}} g' : [o_0, \cdots, (\mathtt{W}, k_2, v'), (\mathtt{W}, k_1, v)], l'' : [o'_0, \cdots, o'_m, (\mathtt{W}, k_2, v'), (\mathtt{W}, k_1, v)]. \end{aligned}$$

Similar to Koskinen et al. [2010], operations must satisfy left mover property when they are pushed to the shared log. They were able to model database and ADTs by instantiating appropriate operations. This means o can be, for example, operations like `Enqueue` or `Dequeue`. Their global log g tracks the history of the database, which is similar to our kv-store tracking the client-observable history. Each client in their semantics has local log, which is a subset of the global store. This is similar to our client partial view. They use the semantics to prove implementations of serialisability. However, it is unknown if their method can be generated to capture consistency models weaker than serialisability.

Apart from transactions in distributed databases, Doherty et al. [2013] proposed an operational semantics using I/O automata to define the correctness of transactional memory. An I/O automata is a labelled transition system comprising a set of states, including starting states, and a transition relation labelled with actions. In [Doherty et al., 2013], the states are standard memories, functions from addresses to values, and the states for transactions. The transitions are either external operations such as single read and write operations of transactions, or external operations such as start, abort and commit operations. They defined specific I/O automata which give rise to two new correctness conditions for transactional memory known as TMS1 and TMS2, and used the semantics to prove implementations by proving simulation. Lesani [2014] gave semantics to well-known correctness conditions for transactional memory including opacity [Guerraoui and Kapalka, 2008], TMS1 and TMS2, using execution histories. They verified many transactional memory algorithms using their semantics.

Chapter 3

Overview

We introduce an interleaving operational semantics for describing client-observable behaviours of atomic transactions on distributed key-value stores. The abstract states of the semantics comprise *global centralised key-value stores* (*kv-stores*) and *partial client views*. Our semantics is parametric in the choice of an *execution test*, which gives rise to a consistency model. We motivate our semantics via two intuitive examples, single-counter and multiple-counters. We later will show that these two examples are *robustness* against parallel snapshot isolation and snapshot isolation. An application is *robust* if the behaviours obtained by executing the application under a weak consistency model can be obtained under serialisability. We show that our semantics provide a mid-point for verifying distributed protocols, such as COPS protocol [Lloyd et al., 2011], and proving invariant properties such as robustness. We provide a specific semantics for COPS protocol, which implements causal consistency, and prove that the traces induced by COPS semantics can be refined to causal consistency traces in our semantics. In this chapter, we briefly explain how we verify the COPS protocol. The full details of verifying COPS and Clock-SI are given in Chapter 6. Our semantics can be used to reason about client programs, by proving invariant properties of these programs such as robustness. In contrast to declarative semantics that requires working on the whole history of programs, we are able to prove invariant properties step by step. We prove that, for example, the single-counter is robust against parallel snapshot isolation. We also prove the correctness, despite non-robust, of a lock paradigm under a weak model called update atomic. The full details of client reasoning is given in Chapter 7.

The second application of our operational semantics is to prove invariant properties of transactional libraries (Chapter 7). One such property is *robustness*. A library is robust if for all its client programs `P` and all kv-stores $\mathcal{K}$, if $\mathcal{K}$ is obtained by executing `P` under a weak model, then $\mathcal{K}$ can also be obtained under the stronger model serialisability. That is, the library clients have no observable weak behaviours.

To demonstrate this, we prove the robustness of the single-counter library discussed above against `PSI`, and the robustness of a multi-counter library and the banking library of Alomari

et al. [2008] against SI. We do the latter by proving general invariants that guarantee robustness against our new proposed model WSI which is weaker than SI; hence our robustness proof against WSI implies robustness against stronger models such as SI. As we discuss in Chapter 7, although existing techniques 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.

As well as such robustness properties, we further use our operational semantics to prove *library-specific* properties. In particular, we show that a lock library is correct under PSI, in that it satisfies the *mutual exclusion guarantee*, even though it is not robust. To do this, we simply encode such library-specific guarantees as invariants of the library, and establish them at each step, as described above. By contrast, establishing such library-specific properties using the existing techniques is more difficult. This is because unlike the kv-stores in our operational semantics, existing techniques do not directly record the library *state*; rather, they record full execution traces, making them less amenable for reasoning about such properties.

3.1 Centralised Key-value Stores and Views

Dependency graphs and abstract executions [Adya, 1999; Cerone et al., 2015a; Burckhardt et al., 2012] provide a declarative description for modelling the behaviour of databases under different consistency models. However, these graphs provide little information about how the state of the database evolves throughout the execution of a program. By contrast, we provide an interleaving operational semantics based on an abstract centralised state. The centralised state comprises *a centralised, multi-versioned key-value store*, or *kv-store*, which is *global* in the sense that it contains all the versions written by clients, and *client views of the kv-store*, or *views* which are *partial* in the sense that clients may see different subsets of the versions in the kv-store. Each update is given by either a simple primitive command or an atomic transaction. The atomic transaction steps are subject to an *execution test* which analyses the current kv-store and view of a given client to determine whether the update is allowed by the associated consistency model.

We start with a simple transactional library, $\texttt{Counter}\,(k)$, to introduce our *kv-store* and *views*. Clients of this counter library can manipulate the value of key k via two transactions:

$$\texttt{Inc}\,(\texttt{x}, k) \stackrel{def}{=} [\ \texttt{x} := \texttt{rd}\,(k)\,;\quad \texttt{wt}\,(k, \texttt{x}+1)\ ] \qquad\qquad \texttt{Read}\,(\texttt{x}, k) \stackrel{def}{=} [\ \texttt{x} := \texttt{rd}\,(k)\ ]$$

Command $\texttt{x} := \texttt{rd}\,(k)$ reads the value of key k to local variable x and command $\texttt{wt}\,(k, \texttt{x}+1)$ writes the value of $\texttt{x}+1$ to key k. The code of each operation is wrapped in square brackets, denoting that it must be executed *atomically* as a transaction.

Consider a replicated database, which is a distributed database where data is replicated in

many *sites*[1], also known as *replicas*. For simplicity, we assume that a client only interacts with one replica. For such a replicated database, the correctness of atomic transactions is subtle, depending heavily on the particular consistency model under consideration. For example, consider the following client program

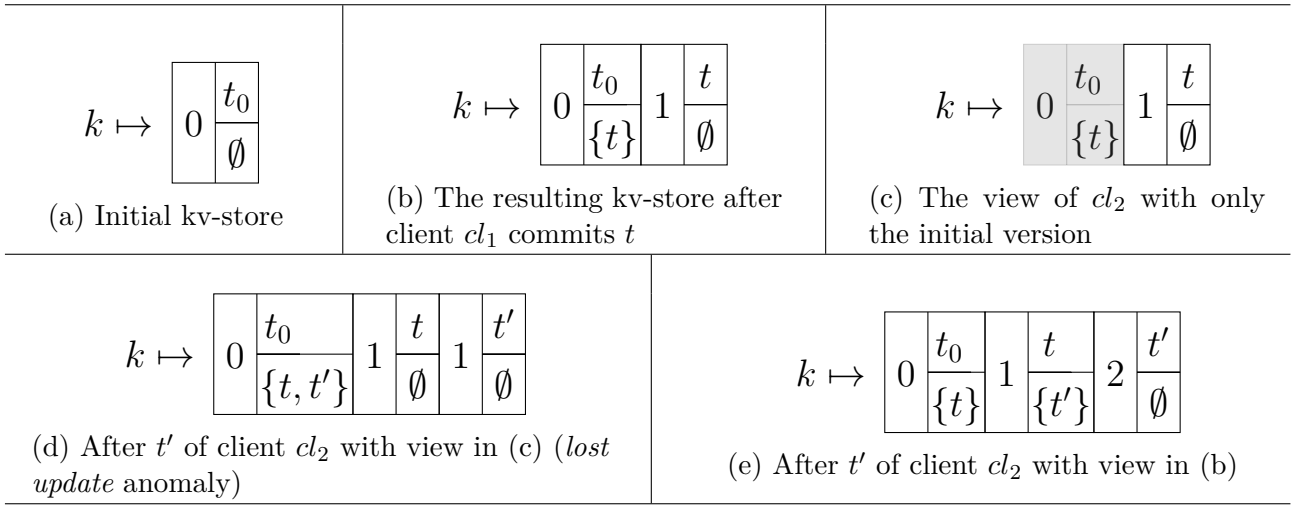
$$\mathtt{P_{LU}} \equiv (\ cl_1 : \mathtt{Inc}\,(\mathtt{x}, k)\ \ ||\ cl_2 : \mathtt{Inc}\,(\mathtt{x}, k)\ )$$

where we assume that the clients cl_1 and cl_2 work on different replicas and the k initially holds value 0 in all replicas. Intuitively, since transactions are executed *atomically*, after both calls to $\mathtt{Inc}\,(k)$ have terminated, the counter should hold the value 2. However, this is not always the case in distributed database or even centralised databases. The *atomicity* of transactions means that transactions take effect in one step. It does not mean that this effect is observable by other clients. Distributed databases use *consistency models* to specify the interaction between transactions[2]. One well-known consistency model is serialisability (SER), where transactions appear to execute in a sequential (serial) order, one after another and a transaction must see all previous transactions. For $\mathtt{P_{LU}}$, the key k must hold value 2 under SER. The implementation of SER comes at a significant performance cost. Therefore, implementers are content with *weak consistency models* [Bailis et al., 2014; Liu et al., 2018; Lloyd et al., 2011; Spirovska et al., 2018; Li et al., 2012; Sovran et al., 2011; Ardekani et al., 2013; Ardekani et al., 2014; Du et al., 2013; Binnig et al., 2014]. For databases especially distributed databases, weak consistency models can increase performance and fault tolerance [Gilbert and Lynch, 2002]. In a weak consistency model, replicas can accept new transactions without full synchronisation between distributed sites. For example, if the replicas provide no synchronisation mechanism for transactions, then it is possible for both clients in $\mathtt{P_{LU}}$ to read the same initial value 0 for k at their distinct replicas, update them to 1, and eventually propagate their updates to other replicas. Consequently, both replicas are unchanged with value 1 for k. This weak behaviour is known as the *lost update* anomaly, which is allowed under the consistency model called *causal consistency* (CC) [Lloyd et al., 2011; Spirovska et al., 2018; Li et al., 2012] but not allowed under *parallel snapshot isolation* (PSI) [Sovran et al., 2011] and *snapshot isolation* (SI) [Berenson et al., 1995].

Let us introduce our global kv-stores and partial client views by showing that we can reproduce the lost update anomaly given by $\mathtt{P_{LU}}$. Our kv-stores are functions mapping keys to lists of versions, where the versions record all the values written to each key together with the meta-data of the transactions that access it: for each version, it contains: (1) the transaction that initially wrote the version; and (2) the set of transactions that read the version. In the $\mathtt{P_{LU}}$ example, for simplicity, the initial kv-store only comprises a single key k, with only one initial version $(0, t_0, \emptyset)$, stating that k holds value 0, the version *writer* is the special initialisation transaction t_0, and the version *reader set* is empty. Fig. 3.1a depicts this initial kv-store, with the version represented as a box sub-divided in three sections: the value 0; the writer t_0; and

[1] we use sites to refer to abstract nodes or servers in distributed systems.

[2] Traditional relational databases often use the term *isolation levels*.



(a) Initial kv-store

(b) The resulting kv-store after client cl_1 commits t

(c) The view of cl_2 with only the initial version

(d) After t' of client cl_2 with view in (c) (*lost update* anomaly)

(e) After t' of client cl_2 with view in (b)

Figure 3.1: Example key-value stores (a, b, d, e) and client views (b, c)

the reader set $\emptyset$.

First, suppose that cl_1 commits `Inc` on Fig. 3.1a. It does this by choosing a fresh transaction identifier, t, and then proceeds with $\texttt{Inc}\,(\texttt{x}, k)$. The transaction reads the initial version of k with value 0 and then writes a new value 1 for k. The resulting kv-store is depicted in Fig. 3.1b, where the reader set of initial version of k has been updated to reflect that it has been read by t and a new version carrying value 1 has been added.

Second, client cl_2 commits `Inc` on Fig. 3.1b. As there are now two versions available for k, we must determine the version from which cl_2 fetches its value, before executing $\texttt{Inc}\,(\texttt{x}, k)$. This is where *client views* come into play. Intuitively, a view of client cl_2 comprises those versions in the kv-store that are *visible* to cl_2: that is, 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 database [Vogels, 2009]. In our example, there are two view candidates for cl_2 when running $\texttt{Inc}\,(\texttt{x}, k)$ on Fig. 3.1b: (1) one containing only the initial version of k depicted in Fig. 3.1c; and (2) the other containing both versions of k depicted in Fig. 3.1b. As we will explain in Section 4.1, views must include the initial version of each key. With view in Fig. 3.1c, client cl_2 chooses a fresh transaction identifier t', reads the initial value 0 and writes a new version with value 1, which yields the kv-store depicted in Fig. 3.1d. Such a kv-store does not contain a version with value 2, despite two increments on k, producing the *lost update* anomaly. For (2), client cl_2 reads the newest value 1 and writes a new version with value 2, which yields the kv-store depicted in Fig. 3.1e.

To avoid undesirable behaviour, such as the lost update anomaly, we use an *execution test* that restricts the possible update at the point of the transaction commit. One such test is to enforce a client to commit a transaction writing to k if and only if its view contains all versions

available in the global state for k. This is captured by the following predicate

$$\texttt{PreClosed}\left(\mathcal{K}, u, \bigcup_{k \in \{k' \mid (\texttt{W},k',\_) \in \mathcal{F}\}} \mathsf{WW}_{\mathcal{K}}^{-1}(k)\right) \tag{UA}$$

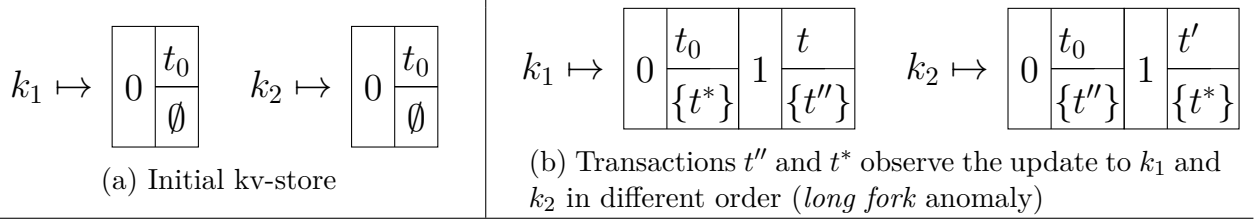
The predicate $\texttt{PreClosed}(\mathcal{K}, u, R)$ asserts that the view u on the kv-store $\mathcal{K}$ must be prefix closed with respect to the relation R. The write-write dependency relation $(t, t') \in \mathsf{WW}_{\mathcal{K}}$, similarly to the relation defined on dependency graphs, means that t' overwrites to a key previous written by t. For example in the kv-store $\mathcal{K}$ presented in Fig. 3.1b, transaction t overwrites the initialisation version, $(t_0, t) \in \mathsf{WW}_{\mathcal{K}}$. The notation $R^?$ denotes the reflexivity of the relation R. Our semantics uses *fingerprints* to capture the effect of transactions. A fingerprint $\mathcal{F}$ is the *read-write set* of a transaction, which contains, for each key, the first read before any subsequent writes and the last write. For example, given the view in Fig. 3.1c, the fingerprint for the increment transaction for client cl_2 is $\{(\texttt{R}, k, 0), (\texttt{W}, k, 1)\}$: that is, the transaction reads the key k with value 0 and writes the key with value 1. This means that if the view u includes versions written by a transaction t, and if $(t', t) \in R$, then u must include versions written by t'. By Eq. (UA), it prevents cl_2 from running $\texttt{Inc}(\texttt{x}, k)$ on the kv-store depicted in Fig. 3.1b with the view depicted in Fig. 3.1c that contains the initial version of k written by t_0. Instead, the view of cl_2 must contain both versions of k, since cl_2 writes a new version to key k and $(t_0, t) \in \mathsf{WW}_{\mathcal{K}}(k)$. Thus Eq. (UA) enforce cl_2 to write a version with value 2 after running $\texttt{Inc}(\texttt{x}, k)$. This particular test corresponds to *update atomic* (UA) that enforces *write-conflict-freedom* of distributed kv-stores: at most one concurrent transaction can write to a key at any one time. We will give all the formal definitions in Chapter 4.

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

$$\mathtt{P}_{\mathtt{LF}} \equiv \begin{pmatrix} cl_1 : [\ \texttt{wt}(k_1, 1)\ ]\,;[\ \texttt{wt}(k_2, 1)\ ] \ \|\ cl_2 : [\ \texttt{x} := \texttt{rd}(k_1)\,;\texttt{y} := \texttt{rd}(k_2)\ ] \\ \qquad\qquad\qquad\qquad\qquad\qquad \|\ cl_3 : [\ \texttt{x} := \texttt{rd}(k_1)\,;\texttt{y} := \texttt{rd}(k_2)\ ] \end{pmatrix}.$$

For simplicity, we assume that the initial kv-store contains two keys (Fig. 3.2a). Suppose that cl_1 executes first the transactions t and t' that updates k_1 and k_2 to values 1 respectively. This results in k_1 and k_2 having two versions with values 0 and 1 each. Client cl_2 executes its transaction t'', using a view that contains both versions of k_1, but only the initial version of k_2 and therefore client cl_2 reads 1 for k_1 and 0 for k_2: that is, cl_2 observes the update of k_1 happening before the increment of k_2. Finally, cl_3 executes its transaction t^* using a view that contains both versions for k_2, but only the initial version of k_1 and therefore client cl_3 reads 0 for k_1 and 1 for k_2, that is, cl_3 observes the increment of k_2 happening before the increment of k_1. This behaviour is known as the *long fork* anomaly (Fig. 3.2b).

The long fork anomaly is disallowed under strong models such as serialisability (SER) and



(a) Initial kv-store

(b) Transactions t'' and t^* observe the update to k_1 and k_2 in different order (*long fork* anomaly)

Figure 3.2: Multiple counter example

snapshot isolation (`SI`), but is allowed under weaker models such as parallel snapshot isolation (`PSI`), causal consistency (`CC`) and update atomic (`UA`). To capture such consistency models and rule out the long fork anomaly as a possible result of $\mathtt{P}_{\mathtt{LF}}$, we must strengthen the execution test associated with the kv-store. For `SER`, we 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, captured by the following predicate

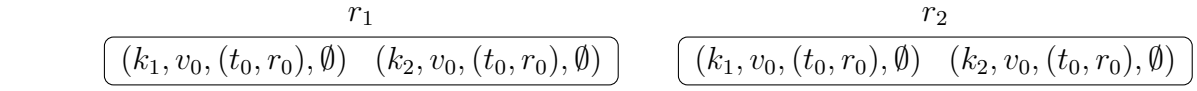
$$\mathtt{PreClosed}\left(\mathcal{K}, u, \mathsf{WW}_{\mathcal{K}}^{-1}\right).$$

For `SI`, the candidate execution test recovers the order in which updates of versions have been observed by different clients:

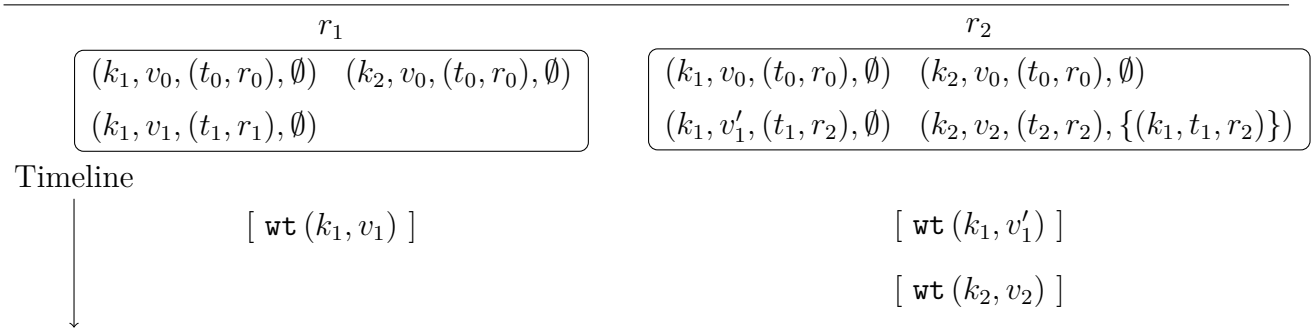
$$\mathtt{PreClosed}\left(\mathcal{K}, u, (\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}})^{+}; \mathsf{RW}_{\mathcal{K}}^{?}\right)$$

where: (1) write-read dependency relation, such as $(t, t'') \in \mathsf{WR}_{\mathcal{K}}$ in Fig. 3.2b, states that a transaction, t, reads a version written by another transaction, t''; (2) session order, SO, determines, for each client cl, the commit order of transactions of cl; and (3) read-write anti-dependency relation, such as $(t^*, t) \in \mathsf{RW}_{\mathcal{K}}$, in Fig. 3.2b, states that a transaction, t^*, reads a version that has been overwritten by another transaction, t. Given this strengthening, client cl_3 must observe the second version of k_1, because $t \xrightarrow{\mathsf{WR}_{\mathcal{K}}} t'' \xrightarrow{\mathsf{RW}_{\mathcal{K}}} t'$ Under such strengthened execution tests for `SER` and `SI`, in the $\mathtt{P}_{\mathtt{LF}}$ example, cl_2 cannot observe 1 for k_2 after observing 0 for k_1, if cl_1 has already established that the increment on k_2 happens after the one of k_1. We will give more detail about the formal definitions of many execution tests for well-known consistency models in Chapter 4.

In Chapter 4 we give formal definition of our semantics and many examples of execution tests and their associated consistency models on kv-stores. In Chapter 5, we show that our definitions of consistency models use execution tests are equivalent to the declarative definitions on abstract executions in the literature. To show the applications of our operational semantics, we use it to verify several distributed protocols (Chapter 6) and prove the invariants of transactional libraries (Chapter 7).



(a) An initial COPS state with two replicas (r_1, r_2) where each replica contains two keys (k_1, k_2) with the initial versions



(b) Clients commit a new version of k_1 carrying value v_1 to replica r_1 and two new versions to r_2 respectively and the new versions in r_1 and r_2 have not yet been synchronised to each other

Figure 3.3: Examples of COPS states

3.2 Application: Verification of Implementations

Kv-stores and views faithfully abstract the state of geo-replicated and partitioned databases, and execution tests provide a powerful abstraction of the synchronisation mechanisms enforced by these databases when committing a transaction. This makes it possible to use our semantics to verify the correctness of distributed database protocols. We demonstrate this by showing that the replicated database, COPS [Lloyd et al., 2011], satisfies causal consistency and the partitioned database, Clock-SI [Du et al., 2013], satisfies snapshot isolation. We present an intuitive account of how we verify the COPS protocol using trace refinement. We refer the reader to Section 6.1 for the full details. In Section 6.2, we apply the same method to verify Clock-SI.

COPS is a fully replicated database, with each replica storing multiple versions of each key as shown in Fig. 3.3b. Each COPS version ν such as the version $(k_1, v_1, (t_1, r_1), \emptyset)$ in Fig. 3.3b, 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 $(d = \emptyset)$, written $\mathsf{DepSetOf}(\nu)$. 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 $d = \mathsf{DepSetOf}(\nu)$ comprises a key and the time-stamp of the version on which ν directly depends. We define the DEP relation, $(t, r) \xrightarrow{\mathsf{DEP}} (t', r')$, to denote that the version identified by (t, r) is included in the dependency set of the version identified by (t', r'). COPS assumes a total order over replica identifiers. As such, versions can be totally ordered lexicographically.

The COPS API provides two operations: $\texttt{put}(k, v)$ for writing to a *single* key and $\texttt{read}(K)$ for atomically reading from *multiple* keys. Operations from a client is processed by a single

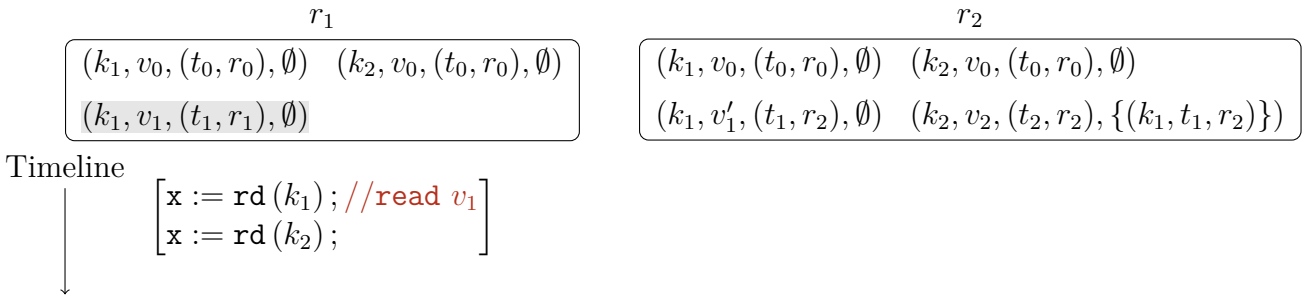
replica. Each client maintains a *context*, which is a set of dependencies tracking the versions the client observes.

We demonstrate how a COPS client cl interacts with a replica through the following example:

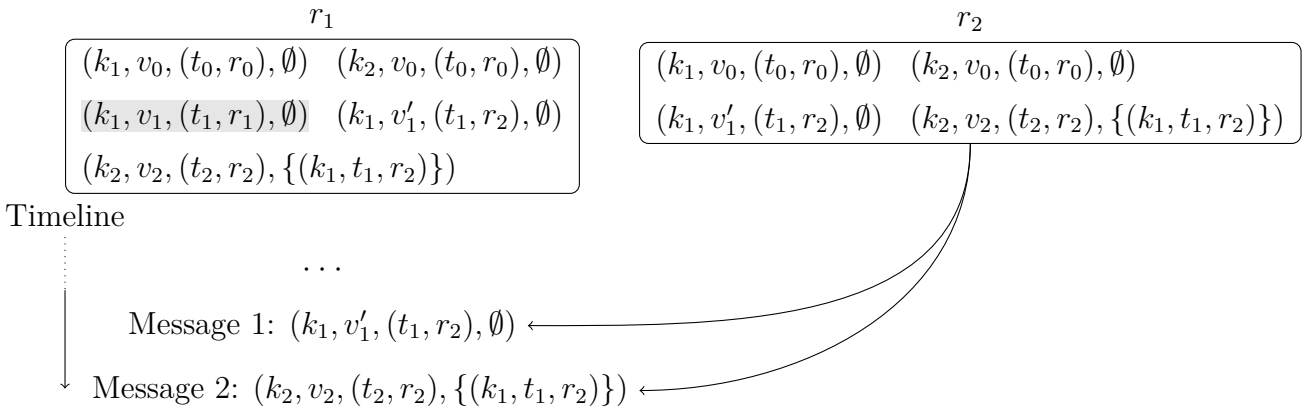
$$\mathtt{P}_{\mathtt{cops-cl}} \equiv cl : \mathtt{put}\,(k_1, v_1)\,;\ \mathtt{read}\,([k_1, k_2])$$

For brevity, we assume that there are two keys, k_1 and k_2, and two replicas, r_1 and r_2, where $r_1 < r_2$ (Fig. 3.3a). We assume that client cl connects to replica r_1 and initialises its local context as $\hat{u} = \emptyset$. To execute its first single-write transaction, cl requests to write v_1 to k_1 by sending the message (k_1, v_1, d), where $d = \hat{u}$, to its associated replica r_1 and awaiting a reply. Upon receiving the message, r_1 produces a monotonically increasing local time t_1, and uses it to install a new version $\nu{=}(k_1, v_1, (t_1, r_1), d)$, as shown in Fig. 3.3b. Note that the dependency set of ν is the cl context ($d = \emptyset$). 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, that is, 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 $\mathsf{DepSetOf}\,(\nu) = \emptyset$. Note that replicas may accept new versions from different clients in parallel.

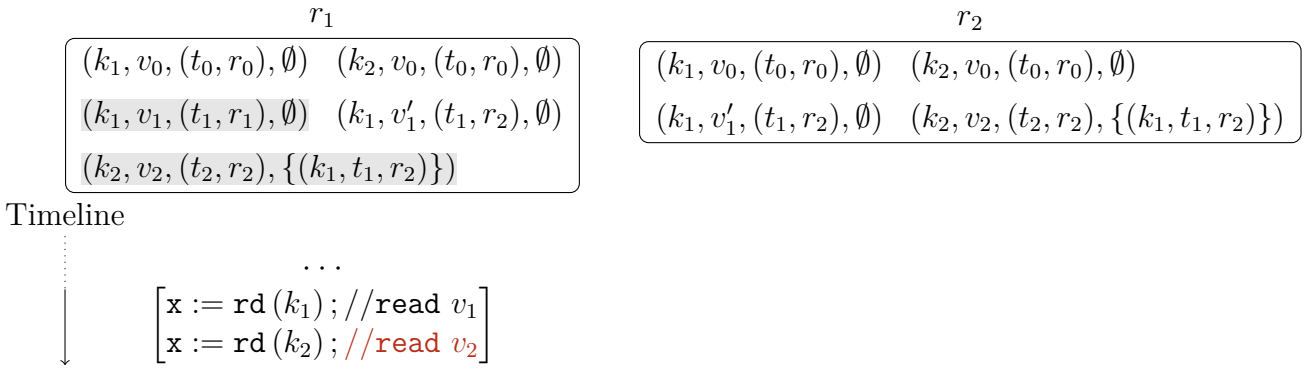
To execute its second multi-read transaction, client cl requests to read from the k_1, k_2 keys by sending the message $\{k_1, k_2\}$ to replica r_1 and awaiting a reply. Upon receiving this message, r_1 builds a DEP-*closed snapshot* (a mapping from $\{k_1, k_2\}$ to values) in two phases as follows. 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 example, Fig. 3.4 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 depicted in Fig. 3.4b, 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 in Fig. 3.4c). As such, the current snapshot for $\{k_1, k_2\}$ are not DEP-closed in the sense that $(k_2, v_2', (t_2, r_2), \{(k_1, t_1, r_2)\})$ depends on a k_1 version with time-stamp (t_1, r_2) which is bigger than (t_1, r_1) for k_1. To remedy this, after the first phase of optimistic reads, r_1 combines (unions) all dependency sets of the versions from the first phase as a *re-fetch set*, and uses it to *re-fetch* the most recent version of each key with the biggest time-stamp from the union of the re-fetch set and the versions from the first phase. For instance, in Fig. 3.4d, 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, and then added to the client context. For their specific implementation, Lloyd et al. [2011] *informally* argue that the snapshot sent to the client is causally consistent. By contrast, we *verify* the COPS protocol using our operational definition of causal consistency on our kv-store.



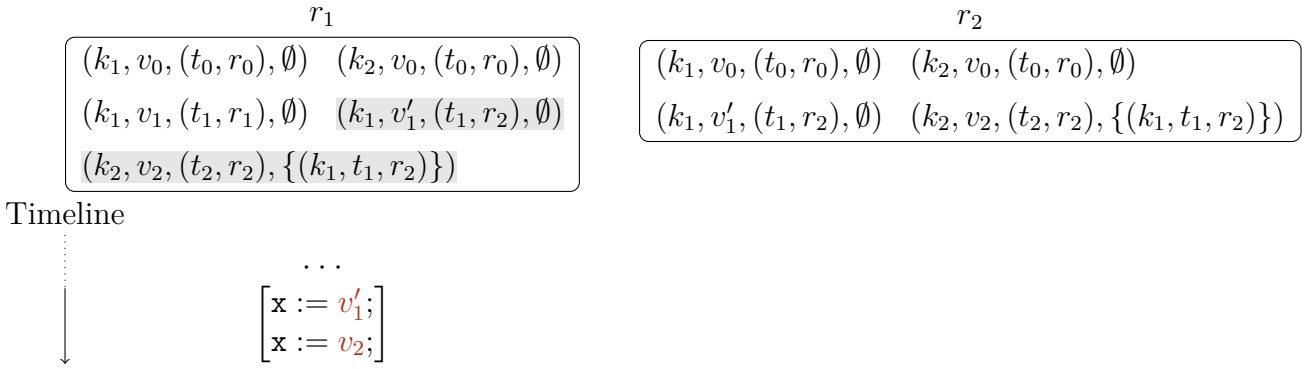
(a) Replica r_1 optimistically fetches the newest version for k_1



(b) Replica r_1 receives two individual synchronisation messages from r_2



(c) Replica r_1 optimistically fetches the newest version for k_2



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

Figure 3.4: Read operation on $[k_1, k_2]$

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

The COPS operational semantics describes transitions over abstract states Θ comprising a set of replicas, a set of client contexts and a program. For instance, the COPS trace that produces Figs. 3.3 and 3.4 is depicted in Fig. 3.5a, 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 (`s`); (3) cl reads version $(\mathtt{R}, k_1, (t_1, r_1))$ from r_1; (4) r_1 receives synchronisation messages (`sync`); (5) cl reads version $(\mathtt{R}, k_2, (t_2, r_2))$ from r_2; (6) cl enters the second phase of the multi-read transaction (`p`); (7) an arbitrary step ι interferes; (8) cl re-fetches the version $(\mathtt{R}, k_1, (t_1, r_2))$ from r_2 and puts it into the snapshot; (9) an arbitrary step ι' interferes; (10) cl puts the version $(\mathtt{R}, k_2, (t_2, r_2))$ into the snapshot; (11) cl reads the values in the snapshot as the final reads for k_1 and k, and commits the multi-read transaction (`e`).

Recall that a multi-read transaction is not executed atomically in the replica, which is captured by multiple read transitions in the trace. For example, the ι and ι' steps in Fig. 3.5a interleave the multi-read transaction from cl. Note that the optimistic reads are not observable to the client and thus it suffices to show that the reads from the second re-fetch phase are atomic. To show this, we *normalise* the trace as follows. For any multi-read transaction, we move all reads from the re-fetch phase to the right towards the return step `e`, so that these reads are no longer interleaved by others. An example of a normalised trace is given in Fig. 3.5b. For any multi-read transaction, the re-fetch phase can only read a version committed before the `p` step. For example, in Fig. 3.5a the multi-read transaction from cl can only read versions in Θ_5. As such, normalising traces does not alter the returned versions of transactions. After normalisation, transactions in the resulting trace can be seen as if executed atomically.

We next show that normalised COPS traces can be refined to traces in our semantics. To do this, we encode the abstract COPS states Θ as configurations in our operational semantics (Fig. 3.5d). We map all COPS replicas to a single kv-store. The writer of a mapped version is uniquely determined by the time-stamp of the corresponding COPS version, while its reader set can be recovered by annotating read-only transactions in the traces: for example, we annotate the read-only transaction from cl in the trace in Fig. 3.5d with t_{rd}. The COPS state in Fig. 3.3b can be encoded as the kv-store depicted in Fig. 3.5c. Similarly, as the context of a client cl identifies the set of COPS versions that cl sees, we can project COPS client contexts to our client views over kv-stores. For example, the contexts of cl before and after committing its second multi-read transaction in $\mathtt{P}_{\mathtt{COPS}}$ is encoded as the client view depicted in Fig. 3.5d.

We finally show that every step in the kv-store trace satisfies $\mathtt{ET}_{\mathtt{CC}}$. Note that existing verification

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

(a) The COPS trace that corresponds Figs. 3.3 and 3.4

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

(b) The normalised COPS trace

$k_1 \mapsto$ | v_0 | (t_0,r_0) / $\emptyset$ | v_1 | (t_1,r_1) / $\emptyset$ | v'_1 | (t_1,r_2) / $\emptyset$ |

$k_2 \mapsto$ | v_0 | (t_0,r_0) / $\emptyset$ | v_2 | (t_2,r_2) / $\emptyset$ |

(c) The kv-store encoding of Fig. 3.3a

$k_1 \mapsto$ | v_0 | (t_0,r_0) / $\emptyset$ | v_1 | (t_1,r_1) / $\emptyset$ | v'_1 | (t_1,r_2) / $\emptyset$ |

$k_2 \mapsto$ | v_0 | (t_0,r_0) / $\emptyset$ | v_2 | (t_2,r_2) / $\emptyset$ |

$\longrightarrow k_1 \mapsto$ | v_0 | (t_0,r_0) / $\emptyset$ | v_1 | (t_1,r_1) / $\emptyset$ | v'_1 | (t_1,r_2) / $\{t_{\mathrm{rd}}\}$ |

$k_2 \mapsto$ | v_0 | (t_0,r_0) / $\emptyset$ | v_2 | (t_2,r_2) / $\{t_{\mathrm{rd}}\}$ |

(d) The step encoding the multi-read transaction in Fig. 3.5a, and views (highlighted) encoding of the client contexts before and after the transaction

Figure 3.5: Examples of a COPS trace, a normalised COPS trace and kv-store trace encoding

techniques [Cerone et al., 2015a; Crooks et al., 2017] 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. One of the key idea of the proof is to show that the *cl* context is always closed to the relation DEP, which contains the relation $R_{\mathsf{CC}} = \mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}$. Recall that SO is session order, and $\mathsf{WR}_{\mathcal{K}}$ is write-read dependency relation. This means that the view u induced by *cl* context satisfy $\texttt{PreClosed}\,(\mathcal{K}, u, R_{\mathsf{CC}})$. we refer the reader to Section 6.1 for the full details.

3.3 Application: Invariants of Client Programs

The second application of our operational semantics is to prove invariant properties of client programs under a weak consistency model. One of such properties is *robustness*. A library is robust if for all its client programs P and all kv-stores $\mathcal{K}$, if $\mathcal{K}$ is obtained by executing P under a weak model, then $\mathcal{K}$ can also be obtained under the stronger model serialisability. That is, the library clients have no observable weak behaviours. With declarative semantics,

Robustness requires checking the shape of the graphs obtained from the whole program. By contrast, we prove robustness via proving an invariant over kv-stores $\mathcal{K}$: that is, the relation $\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{RW}_{\mathcal{K}}$ is acyclic. We prove the robustness of the single counter library discussed above ($\mathtt{P}_{\mathtt{LU}}$) against `PSI`, and the robustness of a multi-counter library and the banking library of Alomari et al. [2008] against `SI`. We do the latter by proving general invariants that guarantee robustness against our new proposed model `WSI` which is weaker than `SI`; hence our robustness proof against `WSI` implies robustness against stronger models such as `SI`.

We further use our operational semantics to prove *library-specific* properties. In particular, we show that a lock pattern is correct under `PSI`, in that it satisfies the *mutual exclusion guarantee*, even though it is not robust. To do this, we simply encode such library-specific guarantees as invariants of the library, and establish them at each step, as described above. By contrast, establishing such library-specific properties using the existing techniques is more difficult, because unlike the kv-stores in our operational semantics, existing techniques do not directly record the library *state*. Instead, they record full execution traces, making them less amenable for reasoning about such properties. All the detail is presented in Chapter 7.

Chapter 4

Operational Semantics

We introduce our interleaving operational semantics for atomic transactions. The abstract machine states of our semantics comprises two key concepts (Section 4.1): (1) *global*, centralised multi-versioning key-value stores (kv-stores), which are abstraction of states of databases, and (2) *partial* client views, which are client observable states. In Section 4.2, we present the rules of our operational semantics. This semantics is parametrised by an *execution test*, that give rise to a consistency model. We then show examples of execution tests for many well-known consistency models in Section 4.3.

Notation. In the definitions, the notation $\mathsf{A} \ni \mathsf{a}$ and $\mathsf{a} \in \mathsf{A}$ denote that the elements of A are ranged over by a and its variants such as $\mathsf{a}', \mathsf{a}_0, \cdots$. The notation $[\mathsf{A}]$ denotes the set of lists over A, and $[\mathsf{a}_0, \cdots, \mathsf{a}_n]$ denotes a list. Given $\mathsf{l} \in [\mathsf{A}]$, the notation $\mathsf{l}_{|i}$, denoting the $(i+1)^{\text{th}}$ element from the list, is defined by:

$$\mathsf{l}_{|i} \stackrel{def}{=} \begin{cases} \mathsf{a}_i & \text{if } \mathsf{l} = [\mathsf{a}_0, \cdots, \mathsf{a}_i, \cdots, \mathsf{a}_n], \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Note that the index starts from 0. Given two lists $\mathsf{l}, \mathsf{l}' \in [\mathsf{A}]$, the notation $\mathsf{l} :: \mathsf{l}'$ denotes the concatenation of the two lists. The notation $|\mathsf{l}|$ denotes the size of the list. The notation $\mathsf{l}[i \mapsto \mathsf{a}]$ denotes the update of the $(i+1)^{\text{th}}$ component to a. For a tuple p, the notation $\mathsf{p}_{|i}$ denotes the $(i+1)^{\text{th}}$ component and $\mathsf{p}[i \mapsto \mathsf{a}]$ denotes the update of the $(i+1)^{\text{th}}$ component to a. The notation $\mathsf{A} \to \mathsf{B}$, $\mathsf{A} \rightharpoonup \mathsf{B}$ and $\mathsf{A} \stackrel{fin}{\rightharpoonup} \mathsf{B}$ denotes the set of total, partial and partial finite functions from A to B respectively. For a function $\mathsf{f} \in \mathsf{A} \to \mathsf{B}$, (similarly for $\mathsf{A} \rightharpoonup \mathsf{B}$ and $\mathsf{A} \stackrel{fin}{\rightharpoonup} \mathsf{B}$), $\mathsf{a} \in \mathsf{A}$ and $\mathsf{b} \in \mathsf{B}$, the notation $\mathsf{f}[\mathsf{a} \mapsto \mathsf{b}]$ denotes the update of the function defined by:

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

4.1 Abstract States: Global Stores and Client Views

We formally define the abstract states of our semantics in this section, which consists of a global, centralised key-value store (kv-store) and several independent partial client views. A kv-store comprises key-indexed lists of versions which record the history of the key with values and meta-data of the transactions that have accessed the version including the writer and readers.

Definition 4.1 (Client and transactional identifiers)**.** *The set of* client identifiers*, $CID \ni cl$, is a countably infinite set. The set of* transaction identifiers*, $\textsc{TxID} \ni t$, is defined by: $\textsc{TxID} \stackrel{def}{=} \{t^n_{cl} \mid cl \in CID \wedge n \in \mathbb{N}\}$. Let $\textsc{TxID}_0 \stackrel{def}{=} \{t_0\} \uplus \textsc{TxID}$.*

Subsets of TxID are ranged over by $T, T', T_1, \cdots$. The transaction identifier t_0 denotes the *initialisation transaction* and t^n_{cl} identifies a transaction committed by client cl with n being used to determine the client session order.

Definition 4.2 (Session order)**.** *The session order,* SO*, is defined by:*

$$\mathsf{SO} \stackrel{def}{=} \left\{ (t^n_{cl}, t^{n'}_{cl}) \;\middle|\; cl \in CID \wedge t^n_{cl}, t^{n'}_{cl} \in \textsc{TxID} \wedge n < n' \right\}.$$

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.

Definition 4.3 (Keys, values and versions)**.** *Assume a countably infinite set of* keys*, $\textsc{Key} \ni k$, and a countably infinite set of* values*, $\textsc{Value} \ni v$, such that $(\textsc{Key} \cup \{v_0\}) \subseteq \textsc{Value}$, where v_0 is the* initialisation value*. The set of* versions*, $\textsc{Version} \ni \nu$, is defined by: $\textsc{Version} \stackrel{def}{=} \textsc{Value} \times \textsc{TxID}_0 \times \mathcal{P}(\textsc{TxID})$. Let $\mathsf{ValueOf}(\nu)$, $\mathsf{WriterOf}(\nu)$ and $\mathsf{ReadersOf}(\nu)$ return the first, second and third components of ν. Given a transaction identifier $t \in \textsc{TxID}$, the notation $t \in \nu$ is defined by: $t \in \{\mathsf{WriterOf}(\nu)\} \cup \mathsf{ReadersOf}(\nu)$.*

Our global centralised key-value stores (kv-stores) keep track of all the history versions for each key. These stores model the overall global state of a system. They provide an abstraction to real-world distributed systems where versions are stored in distributed sites. For example, the COPS replicated database (explained in Section 3.2) can be abstracted to centralised kv-stores.

Definition 4.4 (Kv-stores)**.** *A* kv-store *is a function $\mathcal{K} \in \textsc{Key} \to [\textsc{Version}]$. Given an index $i \in \mathbb{N}$, let $\mathcal{K}(k, i)$ denote the i^{th} element of the list of versions of k, defined by: $\mathcal{K}(k, i) \stackrel{def}{=} \mathcal{K}(k)_{|i}$. Given a transaction t and a kv-store $\mathcal{K}$, the transaction is included in $\mathcal{K}$, written $t \in \mathcal{K}$, if and only if t is either the writer or one of the readers of a version in $\mathcal{K}$: $t \in \mathcal{K} \stackrel{def}{\Leftrightarrow} \exists k, i.\, t \in \mathcal{K}(k, i)$. The* initial kv-store*, $\mathcal{K}_0$, is defined by: $\mathcal{K}_0(k) \stackrel{def}{=} [(v_0, t_0, \emptyset)]$ for all $k \in \textsc{Key}$.*

Definition 4.5 (Well-formed kv-store)**.** *A kv-store $\mathcal{K}$ is well-formed, written* $\texttt{WfKvs}\,(\mathcal{K})$*, if and only if, for any $k \in \textsc{Key}$ and $i, i' \in \mathbb{N}$,*

$$\begin{aligned}
&\mathsf{ReadersOf}\,(\mathcal{K}\,(k,i)) \cap \mathsf{ReadersOf}\,(\mathcal{K}\,(k,i')) \neq \emptyset \\
&\quad \vee \mathsf{WriterOf}\,(\mathcal{K}\,(k,i)) = \mathsf{WriterOf}\,(\mathcal{K}\,(k,i')) \Rightarrow i = i' && (4.1)\\
&\mathsf{ValueOf}\,(\mathcal{K}\,(k,0)) = v_0 && (4.2)\\
&\forall t, t' \in \textsc{TxID}.\, t = \mathsf{WriterOf}\,(\mathcal{K}\,(k,i)) \wedge t' \in \mathsf{ReadersOf}\,(\mathcal{K}\,(k,i)) \Rightarrow (t',t) \notin \mathsf{SO}^? && (4.3)\\
&\forall t, t' \in \textsc{TxID}.\, t = \mathsf{WriterOf}\,(\mathcal{K}\,(k,i)) \wedge t' = \mathsf{WriterOf}\,(\mathcal{K}\,(k,i')) \wedge i < i' \Rightarrow (t',t) \notin \mathsf{SO}^? && (4.4)
\end{aligned}$$

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

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 (Eq. (4.1)). As explained in Chapter 2, snapshot property is a common assumption for distributed databases. We also assume that the version list for each key k has the initial version (the left-most version) carrying the initialisation value v_0, written by the initialisation transaction t_0 (Eq. (4.2)). Finally, we assume that the kv-store agrees with the session order of clients: (1) a client cannot read a version of a key that has been written by a future transaction within the same session (Eq. (4.3)); and (2) the order in which versions are written by a client must agree with its session order (Eq. (4.4)).

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. For example, a COPS client *cl* (explained in Section 3.2) only tracks versions that *cl* has accessed in its client context. We model this *incomplete* information using a *view* of the kv-store which provides a *partial* record of the updates observed by a client.

Definition 4.6 (Views)**.** *Given a well-formed kv-store $\mathcal{K} \in \textit{KVS}$, the set of* views on a kv-store*, $\textsc{ViewOn}\,(\mathcal{K}) \ni u$, is defined by:*

$$\textsc{ViewOn}\,(\mathcal{K}) \stackrel{def}{=} \{u \in \textsc{Key} \to \mathcal{P}\,(\mathbb{N}) \mid \texttt{WfView}\,(\mathcal{K}, u)\}\,,$$

where $\texttt{WfView}\,(\mathcal{K}, u)$ *is defined by: for any $k, k' \in \textsc{Key}$ and $i, i' \in \mathbb{N}$,*

$$\begin{aligned}
&0 \in u\,(k)\,, && (4.5)\\
&i \in u\,(k) \Rightarrow 0 \leq i < |\mathcal{K}\,(k)|\,, && (4.6)\\
&i \in u\,(k) \wedge \mathsf{WriterOf}\,(\mathcal{K}\,(k,i)) = \mathsf{WriterOf}\,(\mathcal{K}\,(k',i')) \Rightarrow i' \in u\,(k')\,. && (4.7)
\end{aligned}$$

Given two views $u, u' \in \textsc{ViewOn}\,(\mathcal{K})$, the order between them is defined by: $u \sqsubseteq u' \stackrel{def}{\Leftrightarrow} \forall k \in$

$\mathsf{Dom}\,(\mathcal{K})\,.\,u\,(k) \subseteq u'(k)$. *The set of* views *is* $\textsc{View} \stackrel{def}{=} \bigcup_{\mathcal{K}\in KVS} \textsc{ViewOn}\,(\mathcal{K})$. *The* initial view, u_0, *is defined by:* $u_0 = \lambda k \in \textsc{Key}.\,\{0\}$.

A well-formed view u on a kv-store $\mathcal{K}$ must contain the initial version of each key (Eq. (4.5)) and the indexes in u must be in range (Eq. (4.6). We require that a client view be *atomic* in that it can see either all or none of the updates of a transaction (Eq. (4.7)).

A view provides an abstraction to client local information. In COPS protocol, each client maintains a context that contains all the versions it has accessed. This information in the context can be captured by our view.

Definition 4.7 (Configurations)**.** *A* configuration, $\Gamma \in \textsc{Conf}$, *is a pair* $(\mathcal{K},\mathcal{U})$ *with a kv-store* $\mathcal{K} \in KVS$, *and a* view environment *on the kv-store* $\mathcal{K}$, *defined by:* $\mathcal{U} \in \textsc{ViewEnvOn}\,(\mathcal{K}) \stackrel{def}{=} \textsc{CID} \stackrel{fin}{\rightharpoonup} \textsc{ViewOn}\,(\mathcal{K})$. *Let* $\textsc{ViewEnv} \stackrel{def}{=} \bigcup_{\mathcal{K}\in KVS} \textsc{ViewEnvOn}\,(\mathcal{K})$. *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 and* $\mathcal{U}_0$ *is an initial view environment defined by: for any client* $cl \in \mathsf{Dom}\,(\mathcal{U})$, $\mathcal{U}\,(cl) = u_0$.

A *configuration* is a pair comprising a kv-store and a function describing the views of a finite set of clients. 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 observes. If $i, i' \in u\,(k)$ and $i < i'$, then cl observes the values carried by versions $\mathcal{K}\,(k,i)$ and $\mathcal{K}\,(k,i')$, and it also sees that the version $\mathcal{K}\,(k,i')$ 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 to capture other resolution policies.

Definition 4.8 (View snapshots)**.** *Given* $\mathcal{K} \in KVS$ *and* $u \in \textsc{ViewOn}\,(\mathcal{K})$, *the* snapshot *of* u *on* $\mathcal{K}$ *is a function,* $\mathsf{Snapshot}\,(\mathcal{K},u) : \textsc{Key} \to \textsc{Value}$, *defined by:*

$$\mathsf{Snapshot}\,(\mathcal{K},u) \stackrel{def}{=} \lambda k.\,\mathsf{ValueOf}\,(\mathcal{K}\,(k,\mathsf{Max}\,(u\,(k))))\,,$$

where $\mathsf{Max}\,(u\,(k))$ *is the maximum element in* $u\,(k)$.

In the COPS protocol, a multi-read transaction constructs a snapshot step by step. In our semantics, the construction steps are captured by Snapshot.

4.2 Operational Semantics

We define the program syntax in Section 4.2.1, the semantics for transactional commands in Section 4.2.2, and the semantics for programs in Section 4.2.3. Our semantics is parametrised by execution tests, which are formally defined in Section 4.2.4.

4.2.1 Programming Language

We assume standard variable stacks which are total functions from variables to values. We assume standard arithmetic expressions and boolean expressions Assume a countably infinite set of variables $\textsc{Var} \ni \mathtt{x}$. Recall the definition of values in Def. 4.3. We assume standard arithmetic expressions and boolean expressions built from values and program variables. For brevity, the evaluation of expressions has no side-effects.

Definition 4.9 (Stacks)**.** *The set of* client-local stacks *or just* stacks*,* $\textsc{Stack} \ni s$*, is defined by* $\textsc{Stack} \stackrel{def}{=} \textsc{Var} \to \textsc{Value}$*.*

Definition 4.10 (Arithmetic and boolean expressions)**.** *The set of* arithmetic expressions*,* $\textsc{Expressions} \ni \mathtt{E}$*, is defined by the grammar:*

$$\mathtt{E} ::= v \mid \mathtt{x} \mid \mathtt{E} + \mathtt{E} \mid \mathtt{E} \times \mathtt{E} \mid \dots$$

The evaluation *of an expression* $\mathtt{E}$ *with respect to stack* s*, written* $[\![\mathtt{E}]\!]_s$*, is defined inductively by:*

$$[\![v]\!]_s \stackrel{def}{=} v \quad [\![\mathtt{x}]\!]_s \stackrel{def}{=} s(\mathtt{x}) \quad [\![\mathtt{E}_1 + \mathtt{E}_2]\!]_s \stackrel{def}{=} [\![\mathtt{E}_1]\!]_s + [\![\mathtt{E}_2]\!]_s \quad [\![\mathtt{E}_1 \times \mathtt{E}_2]\!]_s \stackrel{def}{=} [\![\mathtt{E}_1]\!]_s \times [\![\mathtt{E}_2]\!]_s \quad \dots$$

for $\mathtt{x} \in \textsc{Var}$ *and* $v \in \textsc{Value}$*. The set of* boolean expressions*,* $\textsc{Booleans} \ni \mathtt{B}$*, is defined by the grammar:*

$$\mathtt{B} ::= \mathtt{true} \mid \mathtt{false} \mid \mathtt{E} = \mathtt{E} \mid \mathtt{E} < \mathtt{E} \mid \mathtt{B} \wedge \mathtt{B} \mid \mathtt{B} \vee \mathtt{B} \mid \neg \mathtt{B} \mid \dots$$

The evaluation *of an expression* $\mathtt{B}$ *with respect to stack* s*, written* $[\![\mathtt{B}]\!]_s$*, is defined inductively by:*

$$[\![\mathtt{true}]\!]_s \stackrel{def}{=} \mathtt{true} \quad [\![\mathtt{false}]\!]_s \stackrel{def}{=} \mathtt{false}$$
$$[\![\mathtt{E}_1 = \mathtt{E}_2]\!]_s \stackrel{def}{=} [\![\mathtt{E}_1]\!]_s = [\![\mathtt{E}_2]\!]_s \quad [\![\mathtt{E}_1 < \mathtt{E}_2]\!]_s \stackrel{def}{=} [\![\mathtt{E}_1]\!]_s < [\![\mathtt{E}_2]\!]_s$$
$$[\![\mathtt{B}_1 \wedge \mathtt{B}_2]\!]_s \stackrel{def}{=} [\![\mathtt{B}_1]\!]_s \wedge [\![\mathtt{B}_2]\!]_s \quad [\![\mathtt{B}_1 \vee \mathtt{B}_2]\!]_s \stackrel{def}{=} [\![\mathtt{B}_1]\!]_s \vee [\![\mathtt{B}_2]\!]_s \quad [\![\neg \mathtt{B}]\!]_s \stackrel{def}{=} \neg [\![\mathtt{B}]\!]_s \dots$$

Other unary and binary operations, such as subtraction $-$ for arithmetic expression and greater or equal comparison $\geq$ for boolean expression, are defined analogously.

A *program* $\mathtt{P}$ is a finite partial function from client identifiers to sequential client commands. For clarity, we often write $\mathtt{C}_1 \parallel \dots \parallel \mathtt{C}_n$ as syntactic sugar for a program $\mathtt{P}$ with n clients associated with identifiers $cl_1 \dots cl_n$, where each client cl_i executes $\mathtt{C}_i$. *Client commands* or just *commands*, are built up from atomic transactions of the form of $[\ \mathtt{T}\ ]$ and primitive commands for manipulating stacks.

Definition 4.11 (Programs, client commands and transactional commands)**.** *The set of* programs*,* $\textsc{Progs} \ni \mathtt{P}$*, is defined by:* $\textsc{Progs} \stackrel{def}{=} \textsc{CID} \stackrel{fin}{\rightharpoonup} \textsc{Commands}$*, where the set of* client com-

mands, *Commands* $\ni \mathtt{C}$, *is defined by:*

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

for $\mathtt{x} \in$ *Var,* $\mathtt{E} \in$ *Expressions and* $\mathtt{B} \in$ *Booleans. The set of* transactional commands*,* *Transactions* $\ni \mathtt{T}$*, is defined by the grammar:*

$$\mathtt{T} ::= \mathtt{skip} \mid \mathtt{T}_p \mid \mathtt{T}\,;\mathtt{T} \mid \mathtt{T}+\mathtt{T} \mid \mathtt{T}^* \qquad \mathtt{T}_p ::= \mathtt{x} := \mathtt{E} \mid \mathtt{assume}\,(\mathtt{B}) \mid \mathtt{x} := \mathtt{rd}\,(\mathtt{E}) \mid \mathtt{wt}\,(\mathtt{E},\mathtt{E})$$

Client commands $\mathtt{C}$ comprise $\mathtt{skip}$, primitive commands $\mathtt{C}_p$, atomic transactions $[\ \mathtt{T}\ ]$, and standard compound constructs: sequential composition $\mathtt{C};\mathtt{C}$; non-deterministic choice $\mathtt{C}+\mathtt{C}$; and finite iteration $\mathtt{C}^*$. The $\mathtt{if}\,(\mathtt{B})\ \mathtt{C}_1\ \mathtt{else}\ \mathtt{C}_2$ can be encoded as $(\mathtt{assume}\,(\mathtt{B});\mathtt{C}_1)+(\mathtt{assume}\,(\neg\mathtt{B});\mathtt{C}_2)$. Primitive commands include variable assignment $\mathtt{x} := \mathtt{E}$ and assume statements $\mathtt{assume}\,(\mathtt{B})$ which can be used to encode conditionals. The primitive commands are used for computations based on client-local variables and can hence be invoked without restriction. Transactional commands $\mathtt{T}$ comprise $\mathtt{skip}$, primitive transactional commands $\mathtt{T}_p$, and the standard compound constructs. Primitive transactional commands comprise primitive commands, lookup $\mathtt{x} := \mathtt{rd}\,(\mathtt{E})$ and mutation $\mathtt{wt}\,(\mathtt{E},\mathtt{E})$ used for reading and writing to kv-stores respectively, which can only be invoked as part of an atomic transaction.

4.2.2 Transaction Semantics

In our operational semantics, transactions are executed *atomically*: that is, reduced in one step. This choice simplifies our semantics in that there is no fine-grained interleaving between transactions. It just means that transactions can be treated as executed in one step from the point of individual clients. However, it is still possible for an underlying implementation such as COPS, to access the distributed kv-store in a fine-grained manner. In Section 3.2, we have explained that read-only transactions are executed in a fine-grained method, however, clients do not observe the intermediate steps. In Section 6.1, we give a formal proof that a fine-grained COPS trace is equivalent to a coarse-grained trace in the sense that transactions are executed atomically.

Intuitively, given a configuration $\Gamma = (\mathcal{K},\mathcal{U})$, when a client cl executes a transaction $[\ \mathtt{T}\ ]$, it performs the following steps:

(1) the client constructs an initial *snapshot* σ of $\mathcal{K}$ using its view $\mathcal{U}\,(cl)$ as defined in Def. 4.8;

(2) the client executes $\mathtt{T}$ in isolation over σ accumulating the effects (the reads and writes) of the execution of $\mathtt{T}$; and

(3) the client commits the transaction $[\ \mathtt{T}\ ]$ by incorporating these effects into $\mathcal{K}$.

We explain the semantics for executing $\mathtt{T}$ (step (2)) in this section. In Section 4.2.3, we explain how to construct the initial snapshot (step (1)) and commit the transaction (step (3)). The rules for executing transactional commands is given in Fig. 4.2. The machine state of a transaction consists of a stack s, a transactional snapshot σ that tracks the current value for each key, and a fingerprint $\mathcal{F}$ that tracks the effect of a transaction.

Definition 4.12 (Transactional snapshots)**.** *The set of* transactional snapshots*,* $\textsc{Snapshot} \ni \sigma$*, is defined by:* $\textsc{Snapshot} \stackrel{def}{=} \textsc{Key} \to \textsc{Value}$*.*

Note that the view snapshot $\mathsf{Snapshot}\,(\mathcal{K}, u)$ defined in Def. 4.8, is a transactional snapshot. When the meaning is clear, we call a *transactional snapshot* a *snapshot*.

To capture the effects of executing a transaction $\mathtt{T}$ on a snapshot σ of kv-store $\mathcal{K}$, we identify a *fingerprint* of $\mathtt{T}$ on σ which captures the values that $\mathtt{T}$ reads from σ, and the values that $\mathtt{T}$ writes to σ and intends to commit to $\mathcal{K}$. Execution of a transaction in a given configuration (Def. 4.7) may result in more than one fingerprint due to non-determinism, for example non-deterministic choice $\mathtt{T} + \mathtt{T}$.

Definition 4.13 (Operations and Fingerprints)**.** *Let* $\textsc{Op} \ni o$ *denote the set of* read operations *(*$\mathtt{R}$*) and* write operations *(*$\mathtt{W}$*) defined by:* $\textsc{Op} \stackrel{def}{=} \{(l, k, v) \mid l \in \{\mathtt{R}, \mathtt{W}\} \wedge k \in \textsc{Key} \wedge v \in \textsc{Value}\}$*. Let* ϵ *be the empty operation and* $\textsc{Op}_0 \stackrel{def}{=} \textsc{Op} \uplus \{\epsilon\}$*. A* fingerprint $\mathcal{F}$ *is a set of operations,* $\mathcal{F} \subseteq \textsc{Op}$*. The set of* well-formed fingerprints*,* $\textsc{Fp} \ni \mathcal{F}$*, is defined by:*

$$\textsc{Fp} \stackrel{def}{=} \left\{ \mathcal{F} \;\middle|\; \begin{array}{r} \mathcal{F} \subseteq \textsc{Op} \wedge \forall k \in \textsc{Key}.\, \forall l \in \{\mathtt{R}, \mathtt{W}\}\,.\, \forall v, v' \in \textsc{Value}. \\ \{(l, k, v), (l, k, v')\} \subseteq \mathcal{F} \Rightarrow v = v' \end{array} \right\}.$$

We use the empty operation ϵ to captures primitive commands that has no effect on kv-store. A well-formed fingerprint, $\mathcal{F} \in \textsc{Fp}$, contains at most one read operation and at most one write operation for each key and contains no empty operation ϵ.

Fig. 4.1 presents the rules for transactional commands. The rules for compound transactional commands are standard. The rule TChoice non-deterministically chooses one side. The rule TIter either terminates the loop $\mathtt{T}^*$ reducing to $\mathtt{skip}$, or unwinds one step reducing to $\mathtt{T}\,;\mathtt{T}^*$. A sequence of transactional commands is executed from left to right as expected, modelled by the rules TSeqSkip and TSeq.

The only non-standard rule is TPrimitive, which updates the snapshot and the fingerprint of a transaction: (1) the premise $(s, \sigma) \overset{\mathtt{T}_p}{\rightsquigarrow} (s', \sigma')$ describes how executing $\mathtt{T}_p$ affects the local state (the client stack s and the snapshot σ) of a transaction; and (2) the premise $o = \mathsf{GetOp}\,(s, \sigma, \mathtt{T}_p)$ identifies the operation associated with $\mathtt{T}_p$, which can be a read, a write or an empty operation. The fingerprint combination operation, $\mathcal{F} \lll o$, adds a read/write operation to a fingerprint and ignores the empty operation ϵ.

$\rightsquigarrow$: (Stack × Snapshot × Fp × Transactions) × (Stack × Snapshot × Fp × Transactions)

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

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

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

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

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

Figure 4.1: Operational semantics of transactional commands ($\overset{\mathtt{T}_p}{\rightsquigarrow}$ is defined in Def. 4.14, GetOp is defined in Def. 4.15 and $\lll$ is defined in Def. 4.16)

Definition 4.14 (Transactional local state transition relation)**.** *Given a primitive transactional command* $\mathtt{T}_p \in$ *Transactions, the transition relation over client local stacks and transactional snapshots,* $\overset{\mathtt{T}_p}{\rightsquigarrow} \subseteq$ (*Stack* × *Snapshot*) × (*Stack* × *Snapshot*)*, is defined by:*

$$(s,\sigma)\overset{\mathtt{x:=E}}{\rightsquigarrow}(s\,[\mathtt{x} \mapsto [\![\mathtt{E}]\!]_s]\,,\sigma), \qquad (s,\sigma)\overset{\mathtt{assume(B)}}{\rightsquigarrow}(s,\sigma) \;\textit{if}\; [\![\mathtt{B}]\!]_s = \mathtt{true},$$
$$(s,\sigma)\overset{\mathtt{x:=rd(E)}}{\rightsquigarrow}(s\,[\mathtt{x} \mapsto \sigma\,([\![\mathtt{E}]\!]_s)]\,,\sigma), \qquad (s,\sigma)\overset{\mathtt{wt(E,E')}}{\rightsquigarrow}(s,\sigma\,[[\![\mathtt{E}]\!]_s \mapsto [\![\mathtt{E}']\!]_s]).$$

The assignment, $\mathtt{x} := \mathtt{E}$, evaluates the expression E and assigns the value to the local variable x. The assume command, $\mathtt{assume}\,(\mathtt{B})$, does not affect the stack nor snapshot if the boolean B is evaluated to be true. Otherwise the boolean B is false and there is no transition. The look-up, $\mathtt{x} := \mathtt{rd}\,(\mathtt{E})$, reads from the key $[\![\mathtt{E}]\!]_s$, that is, the evaluation of the expression E, and assign the value associated with the key, $\sigma\,([\![\mathtt{E}]\!]_s)$, to the local variable x. The mutation, $\mathtt{wt}\,(\mathtt{E},\mathtt{E}')$, updates the snapshot, assigning the key $[\![\mathtt{E}]\!]_s$ with a new value $[\![\mathtt{E}']\!]_s$.

Definition 4.15 (GetOp function)**.** *Given a stack* $s \in$ *Stack and a transactional snapshot* $\sigma \in$ *Snapshot, the operation associated with a primitive transactional command* $\mathtt{T}_p \in$ *Transactions is defined by:*

$$\mathsf{GetOp}\,(s,\sigma,\mathtt{x} := \mathtt{E}) \overset{def}{=} \epsilon, \qquad \mathsf{GetOp}\,(s,\sigma,\mathtt{assume}\,(\mathtt{E})) \overset{def}{=} \epsilon,$$
$$\mathsf{GetOp}\,(s,\sigma,\mathtt{x} := \mathtt{rd}\,(\mathtt{E})) \overset{def}{=} (\mathtt{R},[\![\mathtt{E}]\!]_s,\sigma\,([\![\mathtt{E}]\!]_s))\,, \qquad \mathsf{GetOp}\,(s,\sigma,\mathtt{wt}\,(\mathtt{E},\mathtt{E}')) \overset{def}{=} (\mathtt{W},[\![\mathtt{E}]\!]_s,[\![\mathtt{E}']\!]_s)\,.$$

where ϵ *is the empty operation.*

The function $\mathsf{GetOp}\,(s, \sigma, \mathtt{T}_p)$ defines the operation associated with every primitive transactional command $\mathtt{T}_p$: (1) the empty operation (ϵ) for the assignment and assume commands, because these primitive command do not contribute to the fingerprint; (2) a read operation (label $\mathtt{R}$) for look-up; and (3) a write operation (label $\mathtt{W}$) for mutation.

Definition 4.16 (Fingerprint combination operations)**.** *Given the operation definition in Def. 4.13, the fingerprint combination operation,* $\lessdot : \mathcal{P}(\mathit{Op}) \times \mathit{Op}_0 \to \mathcal{P}(\mathit{Op})$*, is defined by:*

$$\mathcal{F} \lessdot (\mathtt{R}, k, v) \stackrel{def}{=} \begin{cases} \mathcal{F} \uplus \{(\mathtt{R}, k, v)\} & \text{if } \forall l \in \{\mathtt{R}, \mathtt{W}\}\,.\,\forall v' \in \mathit{Value}.\,(l, k, v') \notin \mathcal{F} \\ \mathcal{F} & \text{otherwise} \end{cases}$$

$$\mathcal{F} \lessdot (\mathtt{W}, k, v) \stackrel{def}{=} (\mathcal{F} \setminus \{(\mathtt{W}, k, v') \mid v' \in \mathit{Value}\}) \uplus \{(\mathtt{W}, k, v)\}\,,$$

$$\mathcal{F} \lessdot \epsilon \stackrel{def}{=} \mathcal{F}.$$

The fingerprint combination operation, $\mathcal{F} \lessdot o$, accumulates the effect of the operation o to the fingerprint $\mathcal{F}$. A read operation from k is added to a fingerprint $\mathcal{F}$ only if $\mathcal{F}$ has no entry for k, thus only recording the first value read for k (before a subsequent write). In contrast, a write to k is always added to $\mathcal{F}$ by removing the existing writes, thus only recording the last write to k. The $\lessdot$ preserves the well-formedness of fingerprints (Prop. A.1 on page 183).

4.2.3 Command Semantics and Program Semantics

The command semantics describes transitions of the form $(\mathcal{K}, u, s)\,, \mathtt{C} \xrightarrow{(cl,\_)}_{\mathtt{ET}} (\mathcal{K}', u', s')\,, \mathtt{C}'$, stating that given the kv-store $\mathcal{K}$, client view u and client-local stack s, the client cl may execute command $\mathtt{C}$ for one step, updating the kv-store to $\mathcal{K}'$, the client-local stack to s', and the command to its continuation $\mathtt{C}'$. The rules for commands are given in Fig. 4.2. The rules for compound commands are standard. The only non-standard rule is CAtomicTrans.

Each transition step is labelled with the client cl and the information about the transition, which is either of the form $(cl, \bullet)$ denoting that cl executes a primitive command that requires no access to $\mathcal{K}$, or $(cl, u, \mathcal{F})$ denoting that cl commits an atomic transaction with final fingerprint $\mathcal{F}$ under the view u.

Definition 4.17 (Command semantics labels)**.** *The set of kv-store semantics labels,* $\mathit{PLabels} \ni \iota$*, is defined by:*

$$\mathit{PLabels} \stackrel{def}{=} \{(cl, \bullet) \mid cl \in \mathit{CID}\} \uplus \{(cl, u, \mathcal{F}) \mid \exists \mathcal{K} \in \mathit{KVS}.\, cl \in \mathit{CID} \wedge \mathcal{F} \in \mathit{Fp} \wedge u \in \mathit{ViewOn}(\mathcal{K})\}$$

The semantics is parametric in the choice of *execution test* $\mathtt{ET}$, which is used to generate the *consistency model* on kv-stores under which a transaction can execute. In Chapter 5, we give

$$\rightarrow_{\mathtt{ET}} : (\text{KVS} \times \textsc{View} \times \textsc{Commands}) \times \text{PLabels} \times (\text{KVS} \times \textsc{View} \times \textsc{Commands})$$

CAtomicTrans

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

CPrimitive

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

CChoice

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

CIter

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

CSeqSkip

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

CSeq

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

Figure 4.2: Operational semantics of sequential client commands parametrised by ET (Snapshot, NextTxID and UpdateKV are defined in Defs. 4.8, 4.19 and 4.20, and $\overset{\mathtt{C}_p}{\rightsquigarrow}$ is defined in Def. 4.18)

many examples of execution tests for well-known consistency models. In Section 5.5, we prove that our definitions of consistency models using execution tests are equivalent to the existing declarative definitions of consistency models.

In Fig. 4.2, the rule for primitive commands, CPrimitive, depends on the transition relation, $\overset{\mathtt{C}_p}{\rightsquigarrow} \subseteq \textsc{Stack} \times \textsc{Stack}$, that simply describes how the primitive command $\mathtt{C}_p$ affects a client local stack. As the rule CPrimitive only changes the client local state, this rule is labelled with $(cl, \bullet)$.

Definition 4.18 (Primitive command transition relation)**.** *Given a primitive command* $\mathtt{C}_p \in$ *Commands, the* transition relation *over client local stacks,* $\overset{\mathtt{C}_p}{\rightsquigarrow} \subseteq$ *Stack* $\times$ *Stack, is defined by:*

$$s \overset{\mathtt{x:=E}}{\rightsquigarrow} s\,[\mathtt{x} \mapsto [\![\mathtt{E}]\!]_s]\,, \qquad\qquad s \overset{\mathtt{assume(B)}}{\rightsquigarrow} s \text{ if } [\![\mathtt{B}]\!]_s = \mathtt{true}.$$

Fig. 4.2 gives the operational semantics of commands. The rules for compound commands are standard. Rules CChoice, CIter and CSeqSkip are associated with primitive command label $(cl, \bullet)$ because they only affect the commands. The label for CSeq is the same as the premise.

The only interesting rule is the CAtomicTrans rule, which describes the execution of an atomic transaction under the execution test $\texttt{ET}$. The first premise, $u \sqsubseteq u''$, states that the current client view u of the executing command may be advanced to a newer view u''. This premise captures that a client, prior to committing a transaction, receives synchronisation messages. Given the new view u'', the transaction obtains the snapshot σ of the kv-store $\mathcal{K}$, and executes $\texttt{T}$ locally to completion ($\texttt{skip}$), updating the stack to s' while accumulating the fingerprint $\mathcal{F}$. These steps are given by the second, $\sigma = \mathsf{Snapshot}\,(\mathcal{K}, u'')$, and the third premise, $(s, \sigma, \emptyset)\,, \texttt{T} \;\rightsquigarrow^*\; (s', \sigma', \mathcal{F})\,, \texttt{skip}$, of CAtomicTrans. Note that the resulting snapshot σ' is ignored as the effect of the transaction is recorded in the fingerprint $\mathcal{F}$. This is because we focus on consistency models satisfying the snapshot property, hence intermediate steps of a transaction are not observable to other transactions. Prior to commit, the $\texttt{CanCommit}_{\texttt{ET}}\,(\mathcal{K}, u'', \mathcal{F})$ premise ensures that under the execution test $\texttt{ET}$, the final fingerprint $\mathcal{F}$ of the transaction is compatible with the (original) kv-store $\mathcal{K}$ and the client view u'', which is used to take the initial snapshot, and thus the transaction *can commit*. Note that the $\texttt{CanCommit}$ check is parametrised by the execution test $\texttt{ET}$. This is because the conditions checked upon committing depend on the consistency model under which the transaction is to commit. In Section 4.3, we define $\texttt{CanCommit}$ for several execution tests associated with well-known consistency models.

Now the client cl is ready to actually commit the transaction. This results in a new kv-store $\mathcal{K}'$ and the client view u'' *shifts* to a new view u':

(1) pick a fresh transaction identifier $t \in \mathsf{NextTxID}\,(cl, \mathcal{K})$ that is greater than any previously used identifiers;

(2) compute the new kv-store via $\mathcal{K}' = \mathsf{UpdateKV}\,(\mathcal{K}, u'', \mathcal{F}, t)$; and

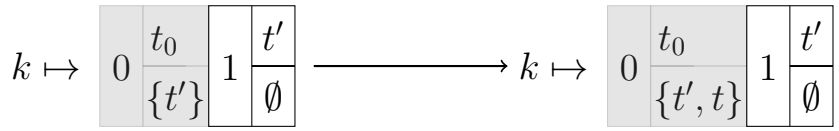
(3) shift the view to u', checking if it is permitted under execution test $\texttt{ET}$ using predicate $\texttt{ViewShift}_{\texttt{ET}}\,(\mathcal{K}, u'', \mathcal{K}', u')$.

Definition 4.19 (Fresh transaction identifiers)**.** *Given a kv-store $\mathcal{K} \in \mathit{KVS}$, the set of* next available transaction identifiers *for a client cl, written $\mathsf{NextTxID}\,(cl, \mathcal{K})$, is defined by:*

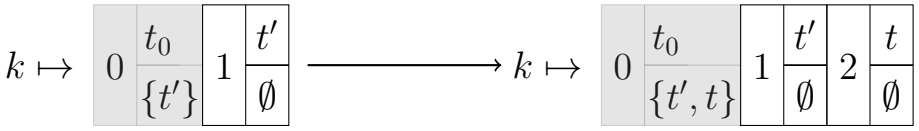
$$\mathsf{NextTxID}\,(cl, \mathcal{K}) \stackrel{def}{=} \{t^n_{cl} \mid t^n_{cl} \in \mathit{TxID} \wedge \forall m \in \mathbb{N}.\, \forall t^m_{cl} \in \mathit{TxID}.\, t^m_{cl} \in \mathcal{K} \Rightarrow m < n\}\,.$$

The fresh transaction identifier for a client cl must be strictly greater than all existing identifiers in the kv-store for the same client cl. This ensures that SO can be determined by the transaction identifiers. Note that the next transaction can be annotated by any number that is bigger than any previous ones, instead of simply adding by ONE. This is useful when we verify implementation protocols in Chapter 6: many implementation protocols use relative times, which do not increase by one.

Definition 4.20 (Kv-store update)**.** *Given a kv-store $\mathcal{K} \in \mathit{KVS}$, a view $u \in \mathit{ViewOn}(\mathcal{K})$, a*



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



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

Figure 4.3: An example of UpdateKV

fingerprint $\mathcal{F} \in \textsc{Fp}$ and a fresh transaction identifier $t \in \textsc{TxID}$, the new kv-store is defined by:

$$\begin{aligned}
\mathsf{UpdateKV}\,(\mathcal{K}, u, \emptyset, t) &\stackrel{def}{=} \mathcal{K}, \\
\mathsf{UpdateKV}\,(\mathcal{K}, u, \{(\mathtt{R}, k, v)\} \uplus \mathcal{F}, t) &\stackrel{def}{=} \mathtt{let}\ i = \mathsf{Max}_<\,(u\,(k))\ \ \mathtt{and}\ \ (v, t', T) = \mathcal{K}\,(k, i)\ \ \mathtt{in} \\
&\qquad \mathtt{let}\ \mathcal{V} = \mathcal{K}\,(k)\,[i \mapsto (v, t', T \cup \{t\})]\ \ \mathtt{in} \\
&\qquad\qquad \mathsf{UpdateKV}\,(\mathcal{K}\,[k \mapsto \mathcal{V}]\,, u, \mathcal{F}, t)\,, \\
\mathsf{UpdateKV}\,(\mathcal{K}, u, \{(\mathtt{W}, k, v)\} \uplus \mathcal{F}, t) &\stackrel{def}{=} \mathsf{UpdateKV}\,(\mathcal{K}\,[k \mapsto (\mathcal{K}\,(k) :: [(v, t, \emptyset)])]\,, u, \mathcal{F}, t).
\end{aligned}$$

The function $\mathsf{UpdateKV}\,(\mathcal{K}, u, \mathcal{F}, t)$ describes how the fingerprint $\mathcal{F}$ of the transaction t is executed under the view u updates the kv-store $\mathcal{K}$: (1) for each read $(\mathtt{R}, k, v) \in \mathcal{F}$, it adds t to the reader set of the last version of k in u, which corresponds to the last-write-wins resolution policy, depicted in Fig. 4.3a; and (2) for each write $(\mathtt{W}, k, v)$, it appends a new version $(v, t, \emptyset)$ to $\mathcal{K}\,(k)$, depicted in Fig. 4.3b.

The function UpdateKV is well-defined in that for any $\mathcal{K}, u, \mathcal{F}, t, cl$ such that $\mathcal{K}$ and $\mathcal{F}$ are well-formed, $u \in \textsc{ViewOn}\,(\mathcal{K})$ and $t \in \mathsf{NextTxID}\,(\mathcal{K}, cl)$, the resulting kv-store $\mathsf{UpdateKV}\,(\mathcal{K}, u, \mathcal{F}, t)$ is a well-formed kv-store.

Theorem 4.21 (Well-defined UpdateKV)**.** *Given a well-formed kv-store $\mathcal{K} \in \textit{KVS}$, a view on the kv-store $u \in \textsc{ViewOn}\,(\mathcal{K})$, a well-formed fingerprint $\mathcal{F} \in \textsc{Fp}$ and a fresh transaction identifier $t \in \mathsf{NextTxID}\,(\mathcal{K}, cl)$ for a client cl, the new kv-store $\mathcal{K}' = \mathsf{UpdateKV}\,(\mathcal{K}, u, \mathcal{F}, t)$ is a uniquely defined and well-formed kv-store.*

Proof sketch. Intuitively, the fingerprint $\mathcal{F}$ contains at most one read and one write per key, therefore the resulting kv-store must be unique defined. The fresh transaction identifier t is strictly greater than any existing identifiers in $\mathcal{K}$ for the same client cl, and the view is a well-formed view on the kv-store $\mathcal{K}$, therefore the resulting kv-store must be well-formed. The full detail is given in appendix A.1 on page 183. ■

Observe that as with `CanCommit`, the `ViewShift` predicate is parametrised by the execution test `ET`. Again, this is because the conditions checked for shifting the client view depend on the

$$\rightarrow_{\mathtt{ET}} : (\text{KVS} \times \textsc{ViewEnv} \times \textsc{ClientEnvs} \times \textsc{Progs}) \times \textsc{Labels} \times (\text{KVS} \times \textsc{ViewEnv} \times \textsc{ClientEnvs} \times \textsc{Progs})$$

$$\textsc{Prog}\ \frac{u = \mathcal{U}(cl) \qquad s = \mathcal{E}(cl) \qquad \mathtt{C} = \mathtt{P}(cl) \qquad \iota = (cl, \cdots) \qquad (\mathcal{K}, u, s), \mathtt{C} \xrightarrow{\iota}_{\mathtt{ET}} (\mathcal{K}', u', s'), \mathtt{C}'}{(\mathcal{K}, \mathcal{U}, \mathcal{E}), \mathtt{P} \xrightarrow{\iota}_{\mathtt{ET}} (\mathcal{K}', \mathcal{U}[cl \mapsto u'], \mathcal{E}[cl \mapsto s']), \mathtt{P}[cl \mapsto \mathtt{C}']}$$

Figure 4.4: Operational semantics of programs

consistency model. In Section 4.3, together with `CanCommit`, we define `ViewShift` for several execution tests associated with well-known consistency models.

Instead of the CAtomicTrans rule, it would be possible to separate the non-deterministic view shift, $u \sqsubseteq u''$ into a separate rule: that is a client can always non-deterministically advance its view. This choice models client non-deterministically receiving an update from the distributed system. However, many real-world distributed systems only passively respond to client requests. We choose the approach in our CAtomicTrans rule where clients update their views only when they commit a new transaction.

We focus on consistency models that satisfies *snapshot property*. For brevity, our semantics has the *snapshot property* built in:

(1) a view u must be atomic in that it can see either all or none of the update of any committed transaction (Eq. (4.7));

(2) the initial snapshot for a transaction, $\sigma = \mathsf{Snapshot}(\mathcal{K}, u)$, contains the latest observable value for each key, which means that σ contains either all or none of the updates from other transactions; and

(3) the fingerprint of a transaction contains the first read before any write and the last write for each key, which means that no internal operations are observable by other transactions.

The operational semantics of programs is given in Fig. 4.4, comprising rule Prog which captures the execution of a program step $\mathtt{P}$ given configuration, $(\mathcal{K}, \mathcal{U}) \in \textsc{Conf}$, and *client environment*, $\mathcal{E} \in \textsc{ClientEnvs}$. A client environment $\mathcal{E}$ tracks the local stacks for clients.

Definition 4.22 (Client environments)**.** *The set of* client environments*,* $\textsc{ClientEnvs} \ni \mathcal{E}$*, is defined by:* $\textsc{ClientEnvs} \stackrel{def}{=} \mathit{CID} \stackrel{fin}{\rightharpoonup} \textsc{Stack}$*.*

We assume that the domain of the client environment contains (the client view environment) the domain of the program throughout the execution: $\mathsf{Dom}(\mathtt{P}) \subseteq \mathsf{Dom}(\mathcal{E})$ ($\mathsf{Dom}(\mathtt{P}) \subseteq \mathsf{Dom}(\mathcal{U})$, respectively). This ensures that progress of our semantics: each client cl in the program $\mathtt{P}$ has

its local variable stack $\mathcal{E}(cl)$ and view $\mathcal{U}(cl)$. Program transitions are simply defined in terms of the transitions of their constituent client commands. This yields a standard interleaving semantics for transactions of different clients: a client executes a transaction in an atomic step without interference from the other clients. Given an execution test $\mathtt{ET}$, a valid kv-store program trace η of a program $\mathtt{P}$ is a finite trace induced by our operational semantics starting a valid initial state $(\mathcal{K}_0, \mathcal{U}_0), \mathcal{E}_0$: that is, the domains of $\mathcal{U}_0$ and $\mathcal{E}_0$ contains the domain of $\mathtt{P}$.

Definition 4.23 (First and Last functions)**.** *Given any trace* t *of the form* $\mathsf{s}_0 \to \cdots \to \mathsf{s}_n$*, functions* $\mathsf{First}(\mathsf{t})$ *and* $\mathsf{Last}(\mathsf{t})$ *are defined by:* $\mathsf{First}(\mathsf{t}) \stackrel{def}{=} \mathsf{s}_0$ *and* $\mathsf{Last}(\mathsf{t}) \stackrel{def}{=} \mathsf{s}_n$*.*

Definition 4.24 (Program traces and reachable kv-stores)**.** *Given an execution test* $\mathtt{ET}$*, and a program* $\mathtt{P}$*, the set of* program traces*,* $PTraces(\mathtt{ET}, \mathtt{P}, n) \ni \eta$*, is defined by:*

$$PTraces(\mathtt{ET}, \mathtt{P}, 0) \stackrel{def}{=} \left\{ (\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}), \mathtt{P} \;\middle|\; \begin{array}{l} \mathcal{U}_0 \in ViewEnvOn(\mathcal{K}_0) \\ \quad \wedge \mathsf{Dom}(\mathtt{P}) \subseteq \mathsf{Dom}(\mathcal{U}_0) \cup \mathsf{Dom}(\mathcal{E}) \end{array} \right\}$$

$$PTraces(\mathtt{ET}, \mathtt{P}, n+1) \stackrel{def}{=} \left\{ \eta \Rightarrow_{\mathtt{ET}} (\mathcal{K}, \mathcal{U}, \mathcal{E}), \mathtt{P}' \;\middle|\; \begin{array}{l} \eta \in PTraces(\mathtt{ET}, \mathtt{P}, n) \\ \quad \wedge \mathsf{Dom}(\mathtt{P}') \subseteq \mathsf{Dom}(\mathcal{U}) \cup \mathsf{Dom}(\mathcal{E}) \end{array} \right\}$$

and the set of reachable kv-stores*, written* $[\![\mathtt{P}]\!]_{\mathtt{ET}}$*, is defined by:*

$$[\![\mathtt{P}]\!]_{\mathtt{ET}} \stackrel{def}{=} \{\mathcal{K} \mid \exists n.\, \exists \eta \in PTraces(\mathtt{ET}, \mathtt{P}, n)\,.\, ((\mathcal{K}, \_, \_), \_) = \mathsf{Last}(\eta)\}\,.$$

A kv-store $\mathcal{K}$ is *reachable* with respect to a program $\mathtt{P}$, written $\mathcal{K} \in [\![\mathtt{P}]\!]_{\mathtt{ET}}$, if and only if $\mathcal{K}$ is the kv-store of the final state of a valid kv-store program trace of $\mathtt{P}$. This notion defines the expressibility of our semantics. In Section 5.4, we will show our semantics are equally expressive as declarative semantics on abstract executions. This expressibility result is a foundation to prove our definitions of consistency models are equivalent to declarative definitions on abstract executions.

4.2.4 Execution test and ET-traces

We define consistency models for our kv-stores, by introducing the notion of an *execution test* that specifies whether a client is allowed to commit a transaction in a given kv-store. Each execution test induces a consistency model as the set of kv-stores obtained by having clients non-deterministically commit transactions so long as the constraints imposed by the execution test are satisfied. We explore a range of execution tests associated with well-known consistency models in the literature. In this section, we formally define execution tests and traces induced by them. In Section 4.3, we give examples of execution tests for several well-known consistency models and, in Chapter 5, we demonstrate that our definitions of consistency models are equivalent to the established axiomatic definitions over abstract executions [Burckhardt et al., 2012; Cerone et al., 2015a] and dependency graphs [Adya, 1999].

An execution test $\mathtt{ET}$ is a set of tuples of the form $(\mathcal{K}, u, \mathcal{F}, \mathcal{K}', u')$, stating that, under $\mathtt{ET}$, a transaction with the fingerprint $\mathcal{F}$ and the given view u is allowed to commit to the kv-store $\mathcal{K}$, and resulting in the new kv-store $\mathcal{K}'$ and the new view u'.

Definition 4.25 (Execution tests)**.** *Given a* `CanCommit` *and a* `ViewShift` *predicate, an* execution test $\mathtt{ET}$ *is a subset of* $\mathit{KVS} \times \textsc{View} \times \mathit{Fp} \times \mathit{KVS} \times \textsc{View}$*, such that: for all* $(\mathcal{K}, u, \mathcal{F}, \mathcal{K}', u') \in \mathtt{ET}$*,*

$$u \in \textsc{ViewOn}(\mathcal{K}) \wedge u' \in \textsc{ViewOn}(\mathcal{K}') \tag{4.8}$$

$$\forall k \in \mathcal{K}.\, \forall v \in \textsc{Value}.\, (\mathtt{R}, k, v) \in \mathcal{F} \Rightarrow \mathcal{K}(k, \mathsf{Max}_{<}(u(k))) = v \tag{4.9}$$

$$\mathtt{CanCommit}(\mathcal{K}, u, \mathcal{F}) \wedge \mathtt{ViewShift}(\mathcal{K}, u, \mathcal{K}', u')\,. \tag{4.10}$$

Let $\mathtt{CanCommit}_{\mathtt{ET}}$ *and* $\mathtt{ViewShift}_{\mathtt{ET}}$ *denote the* `CanCommit` *and* `ViewShift` *predicates for* $\mathtt{ET}$ *respectively. Let* $\textsc{ExecutionTest} \ni \mathtt{ET}$ *denote the set of all execution tests.*

We use the notation $(\mathcal{K}, u) \xrightarrow{\mathcal{F}}_{\mathtt{ET}} (\mathcal{K}', u')$ as we can think of the $\mathtt{ET}$ tuple as a transition on kv-stores and views. Note that the well-formed conditions enforce *last-write-wins policy* as the value v read in the fingerprint for any key k must match with the value in the last version for the key k in the view u: Eq. (4.9) requires that any read operation of the key k in the fingerprint $\mathcal{F}$ must read from the last version contained in the view u. An execution test $\mathtt{ET}$ is defined using two predicates $\mathtt{CanCommit}_{\mathtt{ET}}$ and $\mathtt{ViewShift}_{\mathtt{ET}}$, as shown in Eq. (4.10), that used in the rule CAtomicTrans in Fig. 4.2 We give several examples of execution tests which give rise to consistency models on kv-stores in Section 4.3.

Given an execution test $\mathtt{ET}$, an $\mathtt{ET}$*-reduction* of the form $(\mathcal{K},\mathcal{U}) \xrightarrow{\iota}_{\mathtt{ET}} (\mathcal{K}',\mathcal{U}')$ is a labelled transition relation over configurations, where the label describes either an arbitrary view shift for a client or the commitment of a fingerprint. The set of $\mathtt{ET}$*-traces* contains all finite sequences of $\mathtt{ET}$-reductions starting from an initial configuration.

Definition 4.26 ($\mathtt{ET}$-reduction and $\mathtt{ET}$-traces)**.** *The set of* $\mathtt{ET}$-reduction labels*,* $\mathit{ETLabels} \ni \iota$*, is defined by:*

$$\mathit{ETLabels} \stackrel{\mathit{def}}{=} \{(cl, u) \mid cl \in \mathit{CID} \wedge u \in \textsc{View}\} \uplus \{(cl, \mathcal{F}) \mid cl \in \mathit{CID} \wedge \mathcal{F} \in \mathit{Fp}\}\,.$$

Given an execution test $\mathtt{ET}$*, the* $\mathtt{ET}$-reduction *is a labelled transition over configurations,* $(\mathcal{K},\mathcal{U}) \xrightarrow{\iota}_{\mathtt{ET}} (\mathcal{K}',\mathcal{U}')$*, defined by:*

(1) if $\iota = (cl, u)$ *for a client* $cl \in \mathit{CID}$ *and a view* $u \in \textsc{ViewOn}(\mathcal{K})$*, then*

$$(\mathcal{K},\mathcal{U}) \xrightarrow{(cl,u)}_{\mathtt{ET}} (\mathcal{K},\mathcal{U}[cl \mapsto u]) \tag{4.11}$$

and $\mathcal{U}(cl) \sqsubseteq u$*; and*

(2) if $\iota = (cl, \mathcal{F})$ *for a fingerprint* $\mathcal{F}$*, then*

$$(\mathcal{K}, \mathcal{U}) \xrightarrow{(cl,\mathcal{F})}_{\mathtt{ET}} (\mathcal{K}', \mathcal{U}\,[cl \mapsto u]) \tag{4.12}$$

where $(\mathcal{K}, \mathcal{U}\,(cl)) \xrightarrow{\mathcal{F}}_{\mathtt{ET}} (\mathcal{K}', u)$ *and* $\mathcal{K}' = \mathsf{UpdateKV}\,(\mathcal{K}, \mathcal{U}\,(cl)\,, \mathcal{F}, t)$ *(defined in Def. 4.20) for* $t \in \mathsf{NextTxID}\,(cl, \mathcal{K})$ *(defined in Def. 4.19).*

The set of ET-traces*,* $\mathit{ETTRACES}\,(\mathtt{ET}) \ni \tau$*, is defined by:*

$$\begin{aligned}
\mathit{ETTRACES}\,(\mathtt{ET}) &\stackrel{def}{=} \bigcup_{n \in \mathbb{N}} \mathsf{ETTracesN}\,(\mathtt{ET}, n) \\
\mathsf{ETTracesN}\,(\mathtt{ET}, 0) &\stackrel{def}{=} \{\Gamma_0 \mid \Gamma_0 \in \mathit{CONF}_0\} \\
\mathsf{ETTracesN}\,(\mathtt{ET}, n+1) &\stackrel{def}{=} \left\{ \tau \xrightarrow{\iota}_{\mathtt{ET}} \Gamma \;\middle|\; \iota \in \mathit{ETLABELS} \right\}
\end{aligned}$$

where the set of initial configurations CONF_0 *is defined in Def. 4.7, and the set of all* traces induced by execution tests *is then defined by:*

$$\mathit{ETTRACES} \stackrel{def}{=} \bigcup_{\mathtt{ET} \in \mathit{EXECUTIONTEST}} \mathit{ETTRACES}\,(\mathtt{ET})\,.$$

A transaction in an ET-trace is either a view-shift (Eq. (4.11)) or a transaction step (Eq. (4.12)). An arbitrary client cl advances its view to a new view u, including more versions, in a view-shift step; this intuitively analogous the client receive synchronisation messages. In a transaction step, a client cl commits a fingerprint $\mathcal{F}$, updating the kv-store to $\mathcal{K}'$ and view to u, if this transition satisfies the execution test ET, that is, $(\mathcal{K}, \mathcal{U}\,(cl)) \xrightarrow{\mathcal{F}}_{\mathtt{ET}} (\mathcal{K}', u)$.

Following similar styles in dependency graphs and abstract executions, in which the consistency models are defined in a way that independent from the programs, the *consistency model* induced by an execution test ET, written $\mathsf{ConsisModel}\,(\mathtt{ET})$, is the set of all kv-stores in the ET-traces.

Definition 4.27 (Consistency models on kv-stores)**.** *The* consistency model *induced by an execution test* ET*, written* $\mathsf{ConsisModel}\,(\mathtt{ET})$*, is defined by*

$$\mathsf{ConsisModel}\,(\mathtt{ET}) \stackrel{def}{=} \{\mathcal{K} \mid \exists \tau \in \mathit{ETTRACES}\,(\mathtt{ET})\,.\,(\mathcal{K}, \_) = \mathsf{Last}\,(\tau)\}\,.$$

In this thesis, we mainly use our operational semantics defined in Section 4.2, and kv-store program traces and reachable kv-store $[\![\mathtt{P}]\!]_{\mathtt{ET}}$ defined in Def. 4.24. However, the definition of ET-trace is an important intermediate step to show that our definition of consistency models using execution tests for kv-stores are equivalent to the declarative definitions on abstract executions (Chapter 5). ET-traces define the expressibility of the execution test ET, which not necessary the same as the expressibility of our operational semantics under the ET, in which the latter takes into account of programs. Also note that in the definition of ET-traces, the view-shifts and

transaction commits are decoupled. In contrast, in our operational semantics (CAtomicTrans in Fig. 4.2), view-shifts (the first premise in CAtomicTrans) and transaction commits are combined in a single step. In Theorem 4.31, we present a non-trivial proof showing that these two are equally expressive in the sense that ET-traces and operational semantics are equally expressive in the sense that any kv-store $\mathcal{K} \in \mathsf{ConsisModel}(\mathtt{ET})$ can be obtained as a result of executing some program P under the execution test ET, and vice versa.

We first introduce normalised ET-traces, in which a transaction step of a client cl is followed by a view-shift step of cl. We then show that every ET-trace τ has an *equivalent normalised* ET-trace τ' and we can construct a program and a trace of this program from the normalised ET-trace and vice versa. Two traces τ and τ' are equivalent, written $\tau \simeq \tau'$ if the kv-stores in the final states of the two traces are the same. A normalised ET-trace τ is a trace where updates and view-shifts for a client must appear together.

Definition 4.28 (ET-trace equivalence)**.** *Two* ET*-traces* $\tau, \tau \in \mathit{ETTraces}$ *are equivalent, written* $\tau \simeq \tau'$*, if and only if,* $\exists \mathcal{K} \in \mathit{KVS}.\, \mathsf{Last}(\tau) = (\mathcal{K}, \_) \wedge \mathsf{Last}(\tau') = (\mathcal{K}, \_)$.

Definition 4.29 (Normalised ET-trace)**.** *A* ET*-trace* $\tau \in \mathit{ETTraces}$ *is* normalised*, written* $\mathtt{NormalisedTrace}(\tau)$*, is defined by:*

$$\mathtt{NormalisedTrace}(\Gamma_0) \overset{def}{\Leftrightarrow} \mathtt{true}$$

$$\mathtt{NormalisedTrace}\left(\tau \xrightarrow{\iota}_{\mathtt{ET}} \Gamma \xrightarrow{\iota'}_{\mathtt{ET}} \Gamma'\right) \overset{def}{\Leftrightarrow} \mathtt{NormalisedTrace}(\tau) \wedge \exists cl \in \mathit{CID}.\, \exists u \in \mathit{View}.\, \exists \mathcal{F} \in \mathit{Fp}.\, (\iota = (cl, u) \Leftrightarrow \iota' = (cl, \mathcal{F})).$$

Theorem 4.30 (Equivalent normal ET-traces)**.** *For any* $\tau \in \mathit{ETTraces}$*, there exists an equivalent normalised trace* τ^**, that is,* $\tau \simeq \tau^*$ *and* $\mathtt{NormalisedTrace}(\tau^*)$.

Proof sketch. We perform the following transformations over the trace τ until it is normalised: for any client cl,

(1) we eliminate the last view-shift(s), if there is no more commit steps from cl afterwards;

(2) we move any intermediate view-shift to the right until it is immediately followed by a commit or another view-shift from cl;

(3) we combine any adjacent view-shifts from cl into one view-shift; and

(4) for any commit step that does not follow a view-shift, we insert an identical view-shift before the commit step.

All steps are guaranteed to terminate since there only are finite steps in the trace, and after each transformation the new trace is one step closer becoming a normalised trace. The full detail is in appendix A.1 on page 185. ■

We now prove that our operational semantics and the ET-traces have the same expressibility, in the sense that both give rise of the same set of kv-stores.

Theorem 4.31 (Equivalent expressibility)**.** *For any* $\mathtt{ET} \in \textit{ExecutionTest}$, $\mathsf{ConsisModel}\,(\mathtt{ET}) = \bigcup_{\mathtt{P} \in \mathit{PROGS}} [\![\mathtt{P}]\!]_{\mathtt{ET}}$*, where* ConsisModel *is defined in Def. 4.27 and* $[\![\mathtt{P}]\!]_{\mathtt{ET}}$ *is defined in Def. 4.24.*

Proof sketch. We prove two directions respectively. For $\mathsf{ConsisModel}\,(\mathtt{ET}) \supseteq \bigcup_{\mathtt{P} \in \mathrm{PROGS}} [\![\mathtt{P}]\!]_{\mathtt{ET}}$, given any program trace η, by induction on the length of η, we construct a ET-trace τ and show that the final kv-stores in η and τ are the same. In each step, if the next transition in η is a local computation, we throw away the local computation. Otherwise, the next transition is a transaction step. We split the view-shifts from the transaction commits, and append these two steps to τ. It is easy to see that the final kv-stores of these two traces are the same.

Consider $\mathsf{ConsisModel}\,(\mathtt{ET}) \subseteq \bigcup_{\mathtt{P} \in \mathrm{PROGS}} [\![\mathtt{P}]\!]_{\mathtt{ET}}$. By Theorem 4.30, assume a normalised ET-trace τ. By induction on the length of τ, two steps per iteration, We construct a program P that matches the fingerprints, and then enforces the same scheduling, so that we have a program trace η with the same final kv-store. In each step, let cl be the next scheduled client and $\mathcal{F}$ be the fingerprint of the new transaction. We now inductively construct a new transaction with the transactional command T, which is then appended to the end of the command of cl. For any read operation in the fingerprint, we append a new look-up command to T. After we convert all read operations, for any write operation, we append a new mutate command to T. Now we append a transaction step, combining the view-shift and fingerprint steps, to the trace η. It is easy to see that the final kv-stores of these two traces are the same. The full proof is given in appendix A.1 on 187. ■

4.3 Consistency Models on Key-value Stores

We give several examples of execution tests which give rise to consistency models on kv-stores. Recall that the *snapshot property* and the *last-write-wins policy* are hard-wired into our semantics. This means that we can only define consistency models that satisfy these two constraints. Although this prevents us expressing interesting consistency models such as *read committed* [Berenson et al., 1995], we are able to express a large variety of consistency models employed by distributed kv-stores, from *read atomic* (RA) to *serialisability* (SER).

Notation. Given two relations $\mathsf{r}, \mathsf{r}' \subseteq \mathsf{A} \times \mathsf{A}$, the notation $\mathsf{a} \xrightarrow{\mathsf{r}} \mathsf{a}'$ denotes $(\mathsf{a}, \mathsf{a}') \in \mathsf{r}$. The notation $\mathsf{r}^?, \mathsf{r}^+, \mathsf{r}^*$ denotes the reflexive, transitive, and transitive and reflexive closures of r. The notation r^{-1} denotes the inverse relation defined by $\mathsf{r}^{-1} \stackrel{def}{=} \{(\mathsf{a}', \mathsf{a}) \mid (\mathsf{a}, \mathsf{a}') \in \mathsf{r}\}$. The notation $\mathsf{r}\,;\mathsf{r}'$ denotes relation composition defined by $\mathsf{r}\,;\mathsf{r}' \stackrel{def}{=} \{(\mathsf{a}, \mathsf{a}') \mid \exists \mathsf{a}'' \in \mathsf{A}.\, (\mathsf{a}, \mathsf{a}'') \in \mathsf{r} \wedge (\mathsf{a}'', \mathsf{a}') \in \mathsf{r}'\}$.

Recall that an execution test ET (Def. 4.25) comprises tuples of the form $(\mathcal{K}, u, \mathcal{F}, \mathcal{K}', u')$ where $\mathtt{CanCommit}_{\mathtt{ET}}\,(\mathcal{K}, u, \mathcal{F})$ and $\mathtt{ViewShift}_{\mathtt{ET}}\,(\mathcal{K}, u, \mathcal{K}', u')$. We define CanCommit and ViewShift

for several consistency models, using a couple of auxiliary definitions. $\texttt{CanCommit}_{\texttt{ET}}(\mathcal{K}, u, \mathcal{F})$ predicate requires the view u be closed with respect to a relation R on transactions in $\mathcal{K}$, in the sense that for any transaction t being the *writer* of a version in u, if $(t', t) \in R^+$, then any version *written* by t' must be included in u. This closure property is captured by `PreClosed`.

Definition 4.32 (Visible transactions and prefix closure)**.** *Given a kv-store $\mathcal{K} \in KVS$ and a view on the kv-store $u \in \textsc{ViewOn}(\mathcal{K})$, the* set of visible transactions *is defined by:*

$$\mathsf{VisTrans}(\mathcal{K}, u) \stackrel{def}{=} \{\mathsf{WriterOf}(\mathcal{K}(k, i)) \mid i \in u(k)\}$$

where WriterOf *is defined in Def. 4.3. Given a binary relation on transactions, $R \subseteq \textsc{TxID} \times \textsc{TxID}$, a view u is* prefix closed *or* closed *with respect to a kv-store $\mathcal{K}$ and the relation R, written* $\texttt{PreClosed}(\mathcal{K}, u, R)$*, if and only if*

$$\mathsf{VisTrans}(\mathcal{K}, u) = \left((R^*)^{-1}(\mathsf{VisTrans}(\mathcal{K}, u)) \setminus \mathsf{ReadOnlyTrans}(\mathcal{K})\right),$$

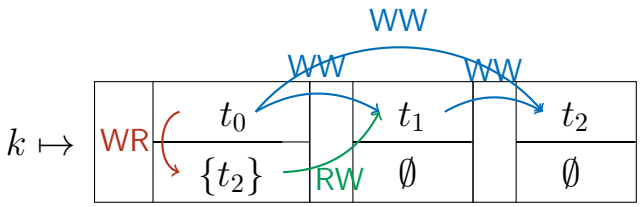
where the set of read-only transactions, $\mathsf{ReadOnlyTrans}(\mathcal{K})$*, is defined by:*

$$\mathsf{ReadOnlyTrans}(\mathcal{K}) \stackrel{def}{=} \{t \mid t \in \mathcal{K} \wedge \forall k \in \textsc{Key}.\, \forall i \in \mathbb{N}.\, t \neq \mathsf{WriterOf}(\mathcal{K}(k, i))\}.$$

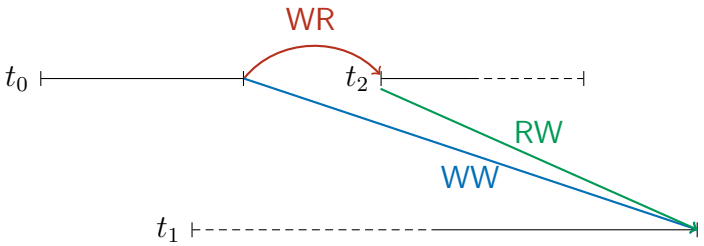
The prefix closure, `PreClosed`, states that if transaction t is visible in u in that a version written by t is included in u, that is, $t \in \mathsf{VisTrans}(\mathcal{K}, u)$, then all transactions t' such that $(t', t) \in R^*$ are also visible in u. The set of prefix closed transactions, $(R^*)^{-1}(\mathsf{VisTrans}(\mathcal{K}, u))$, may contain read-only transactions. However, the read-only transactions have no directly effect on the view in the sense that it does not affect the snapshot induced by the view. Therefore, the predicate `PreClosed` requires that $(R^*)^{-1}(\mathsf{VisTrans}(\mathcal{K}, u)) \setminus \mathsf{ReadOnlyTrans}(\mathcal{K})$ contains the same transactions as $\mathsf{VisTrans}(\mathcal{K}, u)$. Note that a read-only transaction t may *indirectly* affect the view. For example, given a kv-store $\mathcal{K}$ and a view u, if (1) $R = \mathsf{WR} \cup \mathsf{SO}$, (2) $(t'', t) \in \mathsf{WR}$, (3) $(t, t') \in \mathsf{SO}$, and (4) $t' \in \mathsf{VisTrans}(\mathcal{K}, u)$, then $\{t, t''\} \subseteq (R^*)^{-1}(\mathsf{VisTrans}(\mathcal{K}, u))$ and thus $t'' \in (R^*)^{-1}(\mathsf{VisTrans}(\mathcal{K}, u)) \setminus \mathsf{ReadOnlyTrans}(\mathcal{K})$.

We define *dependency relations for kv-stores*, inspired by analogous relations for dependency graphs due to Adya [1999]. The relations are *write-read* (WR), *write-write* (WW) and *read-write* (RW). These three dependency relations, and the session order SO defined in Def. 4.2, are the basic building blocks for defining consistency models using execution tests: the view must be closed with respect to certain combination of these four relations. Note that we specifically use the same names as in dependency graphs (Chapter 5). This is to emphasis the similarity of dependency relations in kv-stores and in dependency graphs.

Definition 4.33 (Dependency relations on kv-stores)**.** *Given a kv-store $\mathcal{K} \in KVS$ and a key $k \in \textsc{Key}$:*



(a) An example of dependency relations on key-value store with values omitted



(b) An example time line contains the starts and commits of transactions, with dashed line being able to stretched

Figure 4.5: Dependency relations on key-value store

(1) the write-read dependency *on key k, written* $\mathsf{WR}_{\mathcal{K}}(k)$*, is defined by:*

$$\mathsf{WR}_{\mathcal{K}}(k) \stackrel{def}{=} \{(t, t') \mid \exists i.\, t = \mathsf{WriterOf}(\mathcal{K}(k, i)) \wedge t' \in \mathsf{ReadersOf}(\mathcal{K}(k, i))\};$$

(2) the write-write dependency *on key k, written* $\mathsf{WW}_{\mathcal{K}}(k)$*, is defined by:*

$$\mathsf{WW}_{\mathcal{K}}(k) \stackrel{def}{=} \{(t, t') \mid \exists i, i'.\, t = \mathsf{WriterOf}(\mathcal{K}(k, i)) \wedge t' = \mathsf{WriterOf}(\mathcal{K}(k, i')) \wedge i < i'\};\ \textit{and}$$

(3) the read-write anti-dependency *on key k, written* $\mathsf{RW}_{\mathcal{K}}(k)$*, is defined by:*

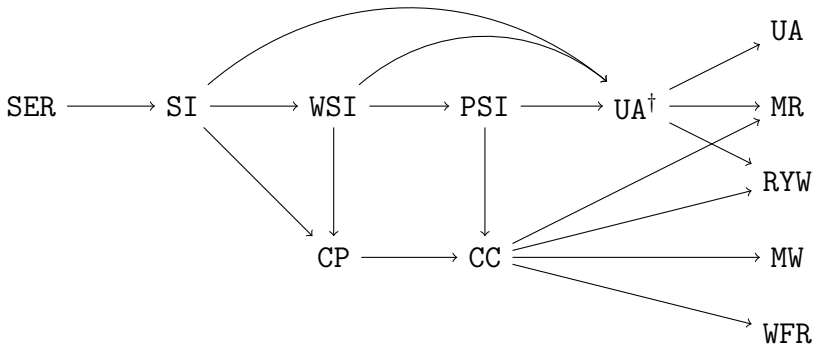
$$\mathsf{RW}_{\mathcal{K}}(k) \stackrel{def}{=} \{(t, t') \mid \exists i, i'.\, t \in \mathsf{ReadersOf}(\mathcal{K}(k, i)) \wedge t' = \mathsf{WriterOf}(\mathcal{K}(k, i')) \wedge i < i' \wedge t \neq t'\}.$$

The write-read, write-write *and* read-write dependencies *on the kv-store $\mathcal{K}$ are then defined by:* $R_{\mathcal{K}} \stackrel{def}{=} \bigcup_{k \in KEY} R_{\mathcal{K}}(k)$ *for* $R \in \{\mathsf{WR}, \mathsf{WW}, \mathsf{RW}\}$.

Fig. 4.5a illustrates an example kv-store and its dependency relations, and Fig. 4.5b is an example time line, if transactions in Fig. 4.5a are executed in a centralised database. The write-read dependency, $(t_0, t_2) \in \mathsf{WR}$, states that transaction t_2 reads a version written by t_0. This means that t_2 *starts* after the commit of t_0, hence t_2 observes effect of t_0, depicted in Fig. 4.5b. The write-write dependency, $(t_0, t_1) \in \mathsf{WW}$, states that transaction t_1 overwrites a version written by t_0. This means that t_1 *commits* after the commit of t_0. However, there is no information about the starts of these two transactions, which means these two transactions may be executed concurrently, as shown in Fig. 4.5b. Note that WW is a total order over all writers of a key. Last, the read-write anti-dependency, $(t_2, t_1) \in \mathsf{RW}$, states that t_2 reads a version that has been over-written by t_1. This means that t_2 starts before the commit of t_1, depicted in Fig. 4.5b,

ET	$\texttt{PreClosed}\,(\mathcal{K}, u, R_{\texttt{ET}})$	$\texttt{ViewShift}\,(\mathcal{K}, u, \mathcal{K}', u')$	Page
$\top$/RA	true	true	57
MR	true	$u \sqsubseteq u'$	58
RYW	true	$\forall t \in \mathcal{K}' \setminus \mathcal{K}. \forall k \in \mathcal{K}'. \forall i.$ $\mathsf{WriterOf}\left(\mathcal{K}'(k,i)\right) \xrightarrow{\mathsf{SO}^?} t \Rightarrow i \in u'(k)$	58
MW	$R_{\texttt{MW}} \stackrel{def}{=} \mathsf{SO} \cap \mathsf{WW}_{\mathcal{K}}$	true	58
WFR	$R_{\texttt{WFR}} \stackrel{def}{=} \mathsf{WR}_{\mathcal{K}}; (\mathsf{SO} \cap \mathsf{RW}_{\mathcal{K}})^?$	true	59
CC	$R_{\texttt{CC}} \stackrel{def}{=} \mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}$	$\texttt{ViewShift}_{\texttt{MR}\cap\texttt{RYW}}\,(\mathcal{K}, u, \mathcal{K}', u')$	59
UA	$R_{\texttt{UA}} \stackrel{def}{=} \bigcup_{(\texttt{W},k,\_)\in\mathcal{F}} \mathsf{WW}_{\mathcal{K}}^{-1}(k)$	true	60
UA†	$R_{\texttt{UA}} \stackrel{def}{=} \bigcup_{(\texttt{W},k,\_)\in\mathcal{F}} \mathsf{WW}_{\mathcal{K}}^{-1}(k)$	$\texttt{ViewShift}_{\texttt{MR}\cap\texttt{RYW}}\,(\mathcal{K}, u, \mathcal{K}', u')$	60
PSI	$R_{\texttt{PSI}} \stackrel{def}{=} R_{\texttt{UA}} \cup R_{\texttt{CC}} \cup \mathsf{WW}_{\mathcal{K}}$	$\texttt{ViewShift}_{\texttt{MR}\cap\texttt{RYW}}\,(\mathcal{K}, u, \mathcal{K}', u')$	61
CP	$R_{\texttt{CP}} \stackrel{def}{=} \mathsf{SO}; \mathsf{RW}_{\mathcal{K}}{}^? \cup \mathsf{WR}_{\mathcal{K}}; \mathsf{RW}_{\mathcal{K}}{}^? \cup \mathsf{WW}_{\mathcal{K}}$	$\texttt{ViewShift}_{\texttt{MR}\cap\texttt{RYW}}\,(\mathcal{K}, u, \mathcal{K}', u')$	62
WSI	$R_{\texttt{WSI}} \stackrel{def}{=} R_{\texttt{UA}} \cup R_{\texttt{CP}}$	$\texttt{ViewShift}_{\texttt{MR}\cap\texttt{RYW}}\,(\mathcal{K}, u, \mathcal{K}', u')$	64
SI	$R_{\texttt{SI}} \stackrel{def}{=} R_{\texttt{UA}} \cup R_{\texttt{CP}} \cup (\mathsf{WW}_{\mathcal{K}}; \mathsf{RW}_{\mathcal{K}})$	$\texttt{ViewShift}_{\texttt{MR}\cap\texttt{RYW}}\,(\mathcal{K}, u, \mathcal{K}', u')$	63
SER	$R_{\texttt{SER}} \stackrel{def}{=} \mathsf{WW}_{\mathcal{K}}^{-1}$	true	63

(a) Execution tests of well-known consistency models, SO is given in Def. 4.2 on page 37.



(b) Hierarchy over execution test in Fig. 4.6a

Figure 4.6: Execution tests

hence, t_2 does not observe the effect of t_1. Note that a transaction might read and write the same, however, RW is irreflexive, hence $(t_2, t_2) \notin \mathsf{RW}$ in Fig. 4.5a. The anti-dependency can be derived from WR and WW in that $(t, t') \in \mathsf{RW} \stackrel{def}{\Leftrightarrow} \exists t''.\, (t'', t) \in \mathsf{WR} \wedge (t'', t') \in \mathsf{WW}$.

Recall that execution tests are defined using ViewShift and CanCommit predicates. We now give several definitions of ViewShift and CanCommit for well-known consistency models in Fig. 4.6a. In Chapter 5, we demonstrate that the associated consistency models on kv-stores correspond to well-known consistency models on abstract executions.

Before we explain all the consistency models, it is straightforward to define a partial order $\sqsubseteq$ over them, depicted in Fig. 4.6b, that is, if $\texttt{ET} \sqsubseteq \texttt{ET}'$, then all the reachable kv-stores by ET are also reachable by ET′. This order states that ET stronger than ET′.

Definition 4.34 (Execution test order)**.** *The order* $\sqsubseteq$ *on execution tests is defined by:*

$$\mathtt{ET} \sqsubseteq \mathtt{ET}' \stackrel{def}{=} \mathsf{ConsisModel}\,(\mathtt{ET}) \subseteq \mathsf{ConsisModel}\,(\mathtt{ET}')$$

where ConsisModel *is defined in Def. 4.27.*

Theorem 4.35 (Execution test order)**.** *Given two execution tests* ET *and* ET′*, for all kv-stores* $\mathcal{K}, \mathcal{K}'$*, views* u, u'*, and fingerprint* $\mathcal{F}$*, if* $\mathtt{CanCommit}_{\mathtt{ET}}\,(\mathcal{K}, u, \mathcal{F}) \Rightarrow \mathtt{CanCommit}_{\mathtt{ET}'}\,(\mathcal{K}, u, \mathcal{F})$ *and* $\mathtt{ViewShift}_{\mathtt{ET}}\,(\mathcal{K}, u, \mathcal{K}', u') \Rightarrow \mathtt{CanCommit}_{\mathtt{ET}'}\,(\mathcal{K}, u, \mathcal{K}', u')$*, then* $\mathtt{ET} \sqsubseteq \mathtt{ET}'$*.*

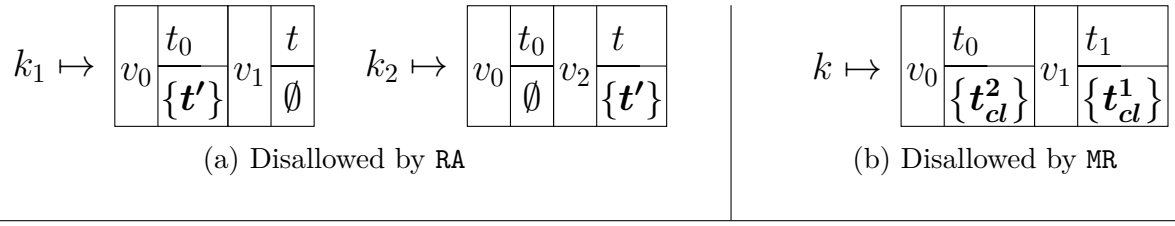
Proof sketch. By the definition of ConsisModel, it is enough to prove that for a trace τ that satisfies ET, this trace τ also satisfies ET′. By Theorem 4.30, we prove this by induction on normalised τ.

- [**Base Case**: $(\mathcal{K}, \mathcal{U})$] It is straightforward that $(\mathcal{K}, \mathcal{U})$ satisfies both ET and ET′.
- [**Base Case**: $\tau \xrightarrow{(cl,u)}_{\mathtt{ET}} (\mathcal{K}, \mathcal{U}') \xrightarrow{(cl,\mathcal{F})}_{\mathtt{ET}} (\mathcal{K}', \mathcal{U}'')$] By inductive hypothesis, we know that τ satisfies ET′. The first view shift is independent from execution test, hence $\tau \xrightarrow{(cl,u)}_{\mathtt{ET}'} (\mathcal{K}, \mathcal{U}')$ satisfies ET′. Let consider the transaction step. By the definition of ET-trace in Def. 4.28, we know $\mathtt{CanCommit}_{\mathtt{ET}}\,(\mathcal{K}, u, \mathcal{F})$ and $\mathtt{ViewShift}_{\mathtt{ET}}\,(\mathcal{K}, u, \mathcal{K}', \mathcal{U}''(cl))$ for some client cl. Therefore, we know $\mathtt{CanCommit}_{\mathtt{ET}'}\,(\mathcal{K}, u, \mathcal{F})$ and $\mathtt{ViewShift}_{\mathtt{ET}'}\,(\mathcal{K}, u, \mathcal{K}', \mathcal{U}''(cl))$, by the hypothesis, which means that $\tau \xrightarrow{(cl,u)}_{\mathtt{ET}'} (\mathcal{K}, \mathcal{U}') \xrightarrow{(cl,\mathcal{F})}_{\mathtt{ET}'} (\mathcal{K}', \mathcal{U}'')$ satisfies ET′. ■

Read atomic ($\top$/RA) This model [Bailis et al., 2014] is a variant of *eventual consistency* [Burckhardt et al., 2012] for atomic transactions. It is the weakest model we can capture, as $\mathtt{CanCommit}_{\mathtt{RA}}$ and $\mathtt{ViewShift}_{\mathtt{RA}}$ are always `true`. We sometimes write $\top$ or $\mathtt{ET}_{\top}$ for this model. In contrast to *atomic* in ACID [Haerder and Reuter, 1983] stating that a transaction *takes effect* in one atomic step, RA states that either none or all effect of a transaction is *observed* by others. RA is also known as *atomic visibility* or *snapshot property*, since in the implementations of RA, a transaction takes an atomic snapshot of the database at the beginning of the transaction, and commits the effects at the end.

Read atomic is hard-wired in our semantics: (1) a view must include all or none of versions written by a transaction; and (2) in the atomic transaction rule, CAtomicTrans, any intermediate steps of a transaction cannot be observed by other transactions. RA prohibits the kv-store in Fig. 4.7a, as transaction t' reads the second version of k_2 carrying value v_2, but not the second version of k_1 carrying value v_1, which are both written by transaction t.

Terry et al. [1994] proposed four *session guarantees*, including *monotonic read* (MR), *read your write* (RYW), *monotonic write* (MW) and *write follows read* (WFR) models However, they only gave informal description of these models in replicated databases. Burckhardt et al. [2014]



(a) Disallowed by RA

(b) Disallowed by MR

Figure 4.7: Anomalies for RA and MR

gave formal definitions on abstract executions using axioms on the visibility relations. Our definitions of these models are correct with respect to theirs. Burckhardt et al. [2014] also proposed two extra session guarantees, called *write follow read in arbitration* and *monotonic write in arbitration*. These two models restrict the arbitration relations on abstract executions. We may not be able to capture these two models, because we do not have the total order information over transactions in our kv-store.

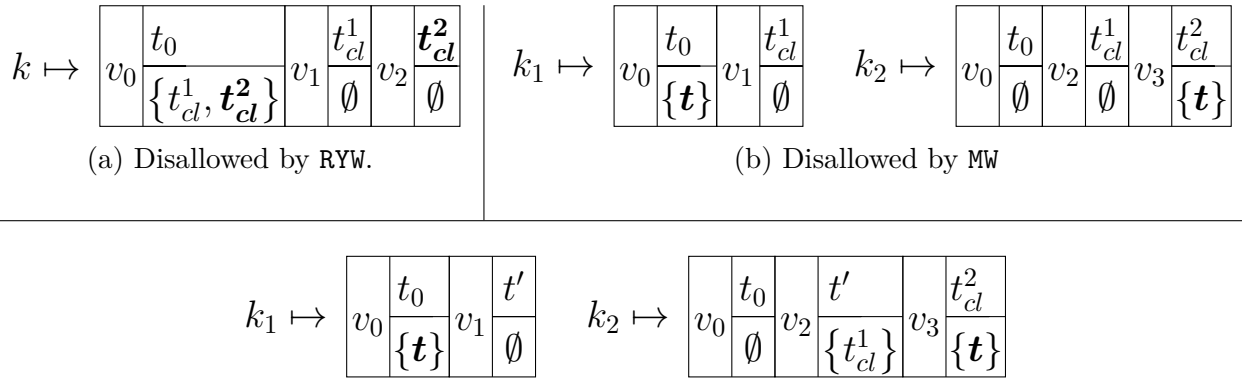
Monotonic read (MR) This model states that after committing a transaction, a client cannot lose information in that it can only see increasingly more versions from a kv-store. This prevents, for example, the kv-store in Fig. 4.7b, since client cl first reads the latest version of k in t^1_{cl}, and then reads the older, initial version of k in t^2_{cl}. As such, the $\texttt{ViewShift}_{\texttt{MR}}$ predicate in Fig. 4.6a ensures that clients can only extend their views, that is, $u \sqsubseteq u'$ for views u, u' before and after committing. When this is the case, clients can then *always* commit their transactions, and thus $\texttt{CanCommit}_{\texttt{MR}}$ is simply defined as `true`.

Read your write (RYW) This model states that a client must always see all the versions written by the client itself. Under RYW the kv-store in Fig. 4.8a is prohibited as the initial version of k holds value v_0 and client cl tries to *update* the value of k twice. For its first transaction t^1_{cl}, it reads the initial value v_0 and then writes a new version with value v_1. For its second transaction t^2_{cl}, it reads the initial value v_0 again and write a new version with value v_2. The $\texttt{ViewShift}_{\texttt{RYW}}$ predicate defined by:

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

rules out this kv-store in Fig. 4.8a by requiring that the client view u', after the commits the transaction t^1_{cl} with $t^1_{cl} \in \mathcal{K}' \setminus \mathcal{K}$, must include t^1_{cl}. When this is the case, clients can always commit their transactions, and thus $\texttt{CanCommit}_{\texttt{RYW}}$ is simply `true`.

Monotonic write (MW) This model states that if a client cl sees a version of a key k that was written by another client cl', then it must see all versions of k that were previously written by cl'. In other words, the view u of the client cl over a kv-store $\mathcal{K}$ must be closed with



(a) Disallowed by `RYW`.

(b) Disallowed by `MW`

(c) Disallowed by `WFR`

Figure 4.8: Anomalies for `RYW`, `MW` and `WFR`

respect to the relation $\mathsf{SO} \cap \mathsf{WW}_{\mathcal{K}}$, before cl can commit a transaction; this is modelled by $\texttt{PreClosed}\,(\mathcal{K}, u, \mathsf{SO} \cap \mathsf{WW}_{\mathcal{K}})$. The resulting view can be any view. Thus $\texttt{ViewShift}_{\texttt{MW}}$ is simply `true`. Monotonic write prohibits the kv-store in Fig. 4.8b, since transaction t reads the third version of k_2 written by t^2_{cl}, but not version written by t^1_{cl} in key k_1, where $(t^1_{cl}, t^2_{cl}) \in \mathsf{SO} \cap \mathsf{WW}_{\mathcal{K}}\,(k_2)$.

Write follow read (`WFR`) This model states that, prior to committing a transaction, if a client cl sees a version on a key k written by some client cl' (possibly equal to cl), then it must also see the versions of the same key k previously read by cl'. This condition is modelled by the predicate $\texttt{PreClosed}\,(\mathcal{K}, u, R_{\texttt{WFR}})$ with $R_{\texttt{WFR}} = \mathsf{WR}_{\mathcal{K}}; (\mathsf{SO} \cap \mathsf{RW}_{\mathcal{K}})^{?}$. Note that if $t \xrightarrow{\mathsf{WR}_{\mathcal{K}}} t' \xrightarrow{(\mathsf{SO} \cap \mathsf{RW}_{\mathcal{K}})^{?}} t''$, then: (1) transactions t', t'' are from the same client; (2) t' reads a version of some k written by t; and (3) later t'' writes a newer version of the same key k. The kv-store in Fig. 4.8c is disallowed by `WFR`, since transaction t reads the third version of key k_2 written by cl, who previous read the second version of key k_2 written by t'. However, transaction t did not read the second version of k_1 also written by t'.

We now give the definitions of well-known consistency models in distributed databases, including *causal consistency* (`CC`) [Shapiro et al., 2011; Lloyd et al., 2011; Burckhardt et al., 2012], *parallel snapshot isolation* (`PSI`) [Sovran et al., 2011; Ardekani et al., 2013], *snapshot isolation* (`SI`) [Berenson et al., 1995] and *serialisability* (`SER`) [Papadimitriou, 1979]. Researchers [Burckhardt et al., 2015; Bernardi and Gotsman, 2016; Cerone et al., 2015a] proposed that the definition of `SI` on abstract executions can be separated into two different consistency models, *update atomic* (`UA`) and *consistent prefix* (`CP`). They also realised that `PSI` can be defined as the conjunction of `UA` and `CC` on abstract executions. Note that, in Fig. 4.6a, the `ViewShift` for `CC`, `CP`, `PSI` and `SI` are defined as the conjunction of the `MR` and `RYW` session guarantees. This is because `MR` and `RYW` are easy to implement in that each client maintains some meta-data about its own history. As explained in Section 2.1, many consistency models are originally defined using specific implementation strategies for tackling certain constraints in distributed

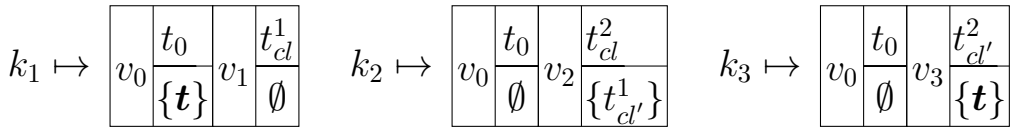


Figure 4.9: Anomaly disallowed by CC

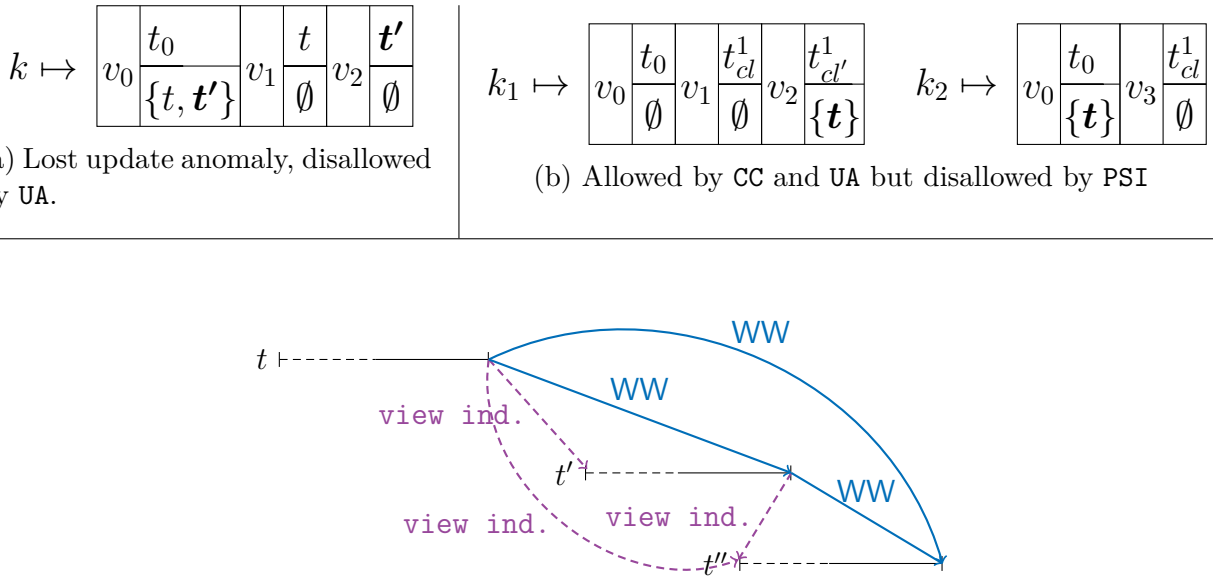
databases.

Causal consistency (CC) This models subsumes the four session guarantees discussed above. As such, the $\texttt{ViewShift}_{\texttt{CC}}$ predicate is defined as $\texttt{ViewShift}_{\texttt{MR}\cap\texttt{RYW}}$. Additionally, CC strengthens the session guarantees by requiring that if a client observes the effect of a transaction t prior to committing a transaction, then it must also observes transactions t' that t observes. If a client observes a version written by t, then t clearly observes all transactions t' from which t directly reads. Moreover, t must observe previous transactions from the same client. This is captured by the $\texttt{CanCommit}_{\texttt{CC}}$ predicate in Fig. 4.6a, defined as $\texttt{PreClosed}\,(\mathcal{K}, u, R_{\texttt{CC}})$ with $R_{\texttt{CC}} \stackrel{def}{=} \mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}$.

For example, the kv-store in Fig. 4.9 is disallowed by CC: the version of key k_3 carrying value v_3 depends on the version of key k_1 carrying value v_1, since $t^1_{cl} \xrightarrow{\mathsf{SO}} t^2_{cl} \xrightarrow{\mathsf{WR}_{\mathcal{K}}} t^1_{cl'} \xrightarrow{\mathsf{SO}} t^2_{cl'}$. However, transaction t must have been committed by a client with view included v_3 but not v_1.

Update atomic (UA) This model disallows concurrent transactions writing to the same key, a property known as *write-conflict freedom*: when two transactions, t and t' write to the same key, one transaction must see the version written by the other. Note that, in distributed systems, the resolution policy only determines the commit order of these two transitions but provide no information about the starts. This means these two transactions may concurrently write to the same key, depicted in Fig. 4.10c. 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. For example, prior to committing, the view of t' includes versions written by t, hence t' must start after t. Similarly, prior to committing, the view of t'' includes versions written by t and t', which ensures t'' starts after t and t'. In other words, its view is closed with respect to the $\mathsf{WW}^{-1}(k)$ relation for all keys k written in the fingerprint $\mathcal{F}$. Recall that a view must include the initial version (left-most version) for each key. UA prevents the kv-store in Fig. 4.10a, known as *lost update anomaly*, as t and t' read initial version of k and update it to v_1 and v_2 respectively. As client views must include the initial versions, once t commits a new version ν with value v_1 to k (the second version in Fig. 4.10a), then t' must include ν in its view as there is a WW edge from the initial version to ν. As such, when t' subsequently updates k, it must read from ν, and not the initial version as depicted in Fig. 4.10a.

This model is originally proposed for the purpose of decomposing of SI, which are defined as a disjointed union of UA and CP (we will explain later). However, UA by itself is not useful since



(a) Lost update anomaly, disallowed by UA.

(b) Allowed by CC and UA but disallowed by PSI

(c) An example of concurrent writes with dashed line being able to stretched

Figure 4.10: Anomalies for UA and PSI

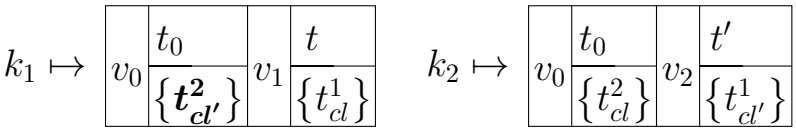
there are no guarantees even on a single session, that is, a client can lose views arbitrary after an update. We proposed $\texttt{UA}^\dagger$ as a strengthen in Fig. 4.6a, which requires MR and RYW alongside the original UA.

Parallel snapshot isolation (PSI) This model can be informally described as:

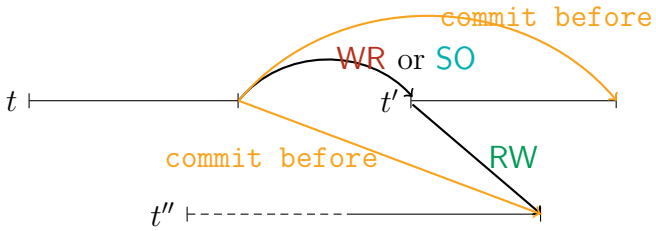
(1) when a transaction t observes the effect of another transaction t, it must also observes the effects of transactions that t' observes;

(2) when two transactions, t and t' write to the same key, one must see the effects from another.

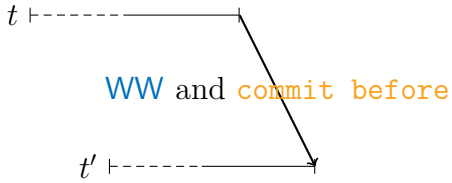
Intuitively, this model is the combination of causal consistency and update atomic. In abstract executions with a total order over transactions, This model is indeed defined as the conjunction of the definitions of CC and UA [Cerone et al., 2015a]. However, we cannot simply define $\texttt{CanCommit}_{\texttt{PSI}}$ as the conjunction of the CanCommit predicates for CC and UA, because in our semantics we do not have the total order. However, the dependency relation provides enough information. The challenge here is the meaning of *observation* between transactions. In CC, the dependence are straightforwardly defined $R_{\texttt{CC}} = \mathsf{WR} \cup \mathsf{SO}$. This is not enough here. Recall that $\texttt{CanCommit}_{\texttt{UA}}$ requires that a transaction writing to a key k must be able to see all previous versions of k. This means that when write-conflict freedom is enforced, a version ν of k *observes* on all previous versions of k, for example Fig. 4.10c. This observation leads us to include write-write dependencies ($\mathsf{WW}_{\mathcal{K}}$) in $R_{\texttt{PSI}}$. Hence, the closure relation for PSI is defined by $R_{\texttt{PSI}} \stackrel{def}{=} R_{\texttt{UA}} \cup R_{\texttt{CC}} \cup \mathsf{WW}_{\mathcal{K}}$. The kv-store in Fig. 4.10b shows an example kv-store that satisfies



(a) Long fork anomaly, disallowed by CP



(b) Commit before relation: WR, SO, WR; RW or WR; RW



(c) Commit before: WW

Figure 4.11: Long fork anomaly and commit before relation

$\texttt{CanCommit}_{\texttt{CC}} \wedge \texttt{CanCommit}_{\texttt{UA}}$, but not $\texttt{CanCommit}_{\texttt{PSI}}$. Assume the commit order is $t^1_{cl}, t^1_{cl'}, t$. The transactions $t^1_{cl}, t^1_{cl'}$ are trivially allowed to commit to the kv-stores under CC, UA and PSI. The transaction t can commit under UA since it does not write any new version, and under CC since there is no causal relation between t^1_{cl} and $t^1_{cl'}$. However, t read the third version of key k_1 written by $t^1_{cl'}$, hence there are edges $t^1_{cl} \xrightarrow{\mathsf{WW}} t^1_{cl'}$, which means if the view used by t includes versions written by $t^1_{cl'}$, the view also need to includes versions written by t^1_{cl} under PSI. Yet t did not read the second version of key k_2 written by t^1_{cl}.

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

In practice, we approximate the order in which transactions commit in a trace that terminates with a final kv-store $\mathcal{K}$ via the $\mathsf{WR}_{\mathcal{K}}, \mathsf{WW}_{\mathcal{K}}, \mathsf{RW}_{\mathcal{K}}$ and SO relations. This approximation is defined by

$$R_{\texttt{CP}} \stackrel{def}{=} \mathsf{WR}_{\mathcal{K}}; \mathsf{RW}_{\mathcal{K}}^{?} \cup \mathsf{SO}; \mathsf{RW}_{\mathcal{K}}^{?} \cup \mathsf{WW},$$

is best understood in terms of an idealised implementation of CP on a *centralised database*, where the snapshot of a transaction is determined at its *start point* and its effects are made visible to future transactions at its *commit point*. With respect to this centralised implementation, first, if $(t, t') \in \mathsf{WR}_{\mathcal{K}}$, then t must commit before t' starts, and hence before t' commits (Fig. 4.11b). Similarly, if $(t, t') \in \mathsf{SO}$, then t commits before t' starts, and thus before t' commits (Fig. 4.11b). Second, recall that $(t', t'') \in \mathsf{RW}_{\mathcal{K}}$ denotes that t'' reads a version that is later overwritten by t'; that is, t' cannot see the write of t'', and thus t' must starts before t'' commits. As such, if t commits before t' starts as $(t, t') \in \mathsf{WR}_{\mathcal{K}}$ or $(t, t') \in \mathsf{SO}$, and t' starts before t''

commits as $(t', t'') \in \mathsf{RW}_{\mathcal{K}}$, then t must commit before t'' commits (Fig. 4.11b). In other words, if $(t, t'') \in \mathsf{WR}_{\mathcal{K}}; \mathsf{RW}_{\mathcal{K}}$ or $(t, t'') \in \mathsf{SO}; \mathsf{RW}_{\mathcal{K}}$, then t commits before t'' (Fig. 4.11b). Last, if $(t, t') \in \mathsf{WW}_{\mathcal{K}}$, then t must commit before t' (Fig. 4.11c). Therefore the relation, $R_{\texttt{CP}}$, approximates the order in which transactions commit. We then define $\texttt{CanCommit}_{\texttt{CP}}$ in Fig. 4.6a by requiring that the client view be closed with respect to $R_{\texttt{CP}}$. Our definition for `CP` via approximating the commit order, is correct with respect to the declarative definition proposed by Cerone et al. [2017].

Consistent prefix disallows the *long fork anomaly* shown in Fig. 4.11a, where clients cl and cl' observe the updates to k_1 and k_2 in different orders. Assuming without loss of generality that $t^2_{cl'}$ was the last transaction committed, then prior to committing its transaction cl' must see the initial version of k_1 with value v_0 and both version of k_2; this is because a view must always include the initial versions and cl' read the second version of k_2 carrying value v_2 in transaction $t^1_{cl'}$. However, since $t \xrightarrow{\mathsf{WR}_{\mathcal{K}}} t^1_{cl} \xrightarrow{\mathsf{SO}} t^2_{cl} \xrightarrow{\mathsf{RW}_{\mathcal{K}}} t'$, then cl' should also see the second version of k_1 with value v_1, leading to a contradiction.

Snapshot isolation (`SI`) This model can be informally described as:

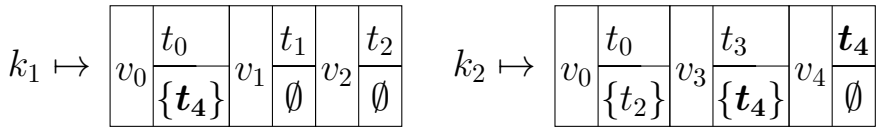
(1) when a transaction t observes the effect of another transaction t', then t must observe transactions that commits before t';

(2) when two transactions, t and t' write to the same key, one must see the effects by another.

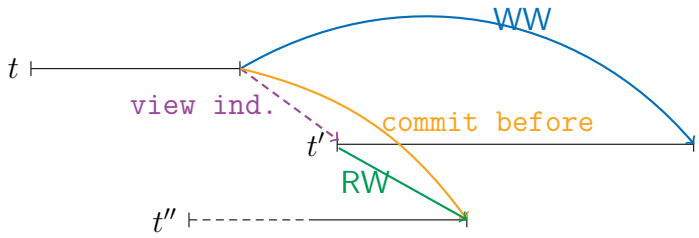
When the total order in which transactions commit is known, *snapshot isolation* (`SI`) can be defined compositionally from `CP` and `UA`. However, as with `PSI`, our semantics does not have the total order. We cannot directly define $R_{\texttt{SI}}$ as $R_{\texttt{CP}} \cup R_{\texttt{UA}}$. Additionally, we include $\mathsf{WW}_{\mathcal{K}}; \mathsf{RW}_{\mathcal{K}}$ in $R_{\texttt{SI}}$. Because when the centralised `CP` implementation (discussed in Section 4.3) is strengthened with *write-conflict freedom*, then a write-write dependency between two transactions t and t' does not only mandate that t commits before t' commits but also before t' starts. Consequently, if $(t, t'') \in \mathsf{WW}_{\mathcal{K}}; \mathsf{RW}_{\mathcal{K}}$, then t must commit before t'' commit (Fig. 4.12b). Therefore in Fig. 4.6a, $\texttt{CanCommit}_{\texttt{SI}}$ is defined as the closure on relation $R_{\texttt{SI}} \stackrel{def}{=} R_{\texttt{UA}} \cup R_{\texttt{CP}} \cup (\mathsf{WW}_{\mathcal{K}}; \mathsf{RW}_{\mathcal{K}})$.

Observe that the kv-store in Fig. 4.12a shows an example kv-store that satisfies $\texttt{CanCommit}_{\texttt{UA}} \wedge \texttt{CanCommit}_{\texttt{CP}}$, but not $\texttt{CanCommit}_{\texttt{SI}}$. In Fig. 4.12a, transaction t_4 must be the last transaction. It must include all versions of k_2 because of write-conflict freedom. However, since $t_1 \xrightarrow{\mathsf{WW}_{\mathcal{K}}} t_2 \xrightarrow{\mathsf{RW}_{\mathcal{K}}} t_3$, transaction t_4 should read the second version of k_1 carrying value v_1, contradicting the fact that t_4 read the initial version of k_1.

Strict serialisability (`SER`) This model is the strongest consistency model in any framework that abstracts from aborted transactions, requiring that transactions execute in a total sequential order. The $\texttt{CanCommit}_{\texttt{SER}}$ thus allows a client to commit a transaction only when



(a) Allowed by UA and CP, and henceforth WSI, but disallowed by SI



(b) Commit before relation: $\mathsf{WW}; \mathsf{RW}$

Figure 4.12: Anomaly for SI and extra commit before relation for SI

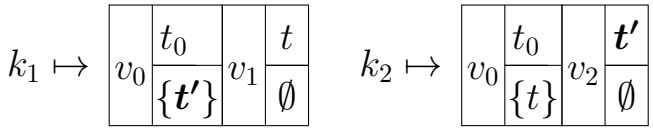


Figure 4.13: Write skew anomaly, disallowed by SER

the client view on the kv-store is complete in that the view is closed with respect to $\mathsf{WW}_{\mathcal{K}}^{-1}$. This requirement prevents the kv-store in Fig. 4.13. Without loss of generality, suppose that t commits before t', then the client committing t' must see the version of k_1 written by t, and thus cannot read the outdated value v_0 for k_1. This example, known as *write skew anomaly*, is allowed by all other execution tests in Fig. 4.6a.

We were surprised to find a new interesting consistency model using our kv-stores. This model, which we call *weak snapshot isolation* (WSI), is defined by combining CP and UA. Under WSI, a client view must be close with respect to relation $R_{\mathtt{WSI}} = R_{\mathtt{CP}} \cup R_{\mathtt{UA}}$, in contrast to SI-relation $R_{\mathtt{SI}} = R_{\mathtt{CP}} \cup R_{\mathtt{UA}} \cup \mathsf{WW}_{\mathcal{K}}; \mathsf{RW}_{\mathcal{K}}$.

WSI is stronger than CP and UA by definition, it therefore forbids all the anomalies forbidden by these consistency models, for example the long fork (Fig. 4.11a) and the lost update (Fig. 4.10a). Moreover, WSI is strictly weaker than SI. As such, WSI allows all SI anomalies such as the write skew (Fig. 4.13), but allows behaviour not allowed under SI such as that in Fig. 4.12a. We can construct a $(\mathtt{CP} \cap \mathtt{UA})$-trace terminating in kv-store $\mathcal{K}$ by executing transactions t_1, t_2, t_3 and t_4 in this order. In particular, t_4 is executed using $u = \{k_1 \mapsto \{0\}, k_2 \mapsto \{0, 1\}\}$ that is allowed by WSI given that $\mathsf{WW}_{\mathcal{K}}; \mathsf{RW}_{\mathcal{K}} \not\subseteq R_{\mathtt{WSI}}$. However, the same trace is not a valid SI-trace as explained in Section 4.3.

To justify this consistency model in full, it would be useful to explore its implementations. Here we focus on the possible benefits of implementing WSI: as WSI is a weaker consistency model

than SI, we believe that WSI implementations would outperform known SI implementations. Nevertheless, the two consistency models are very similar in that many applications that are correct under SI are also correct under WSI. We give an example of such an application in Section 7.2.

We have introduced two key concepts: abstract multi-versioning key-value stores and partial client views (Section 4.1), and our interleaving operational semantics where transactions are executed in atomic steps (Section 4.2). This semantics is parametrised by an execution test that determines if a client with a given view is allowed to commit a transaction (Section 4.2). Our semantics assume last-write-wins resolution policy and snapshot property; many well-known consistency models in distributed systems assume these two properties. We have then given the definitions of many well-known consistency models using execution tests, such as causal consistency, parallel snapshot isolation and snapshot isolation (Section 4.3).

First, our semantics focuses on the abstract states, rather than the interaction between transactions as in declarative semantics, such as [Nagar and Jagannathan, 2018; Adya, 1999; Cerone and Gotsman, 2016; Cerone et al., 2015a], which are graph-based semantics on distributed databases, operationally or declaratively. This immediately allows us to directly reason about the states (in Chapter 7), while in previous work [Nagar and Jagannathan, 2018; Cerone et al., 2015a,b; Cerone and Gotsman, 2016], they can only reason about general invariants that can be encoded as properties on the shapes of the graphs. Second, we provide an operational semantics and operational definitions of consistency models. In contrast to declarative semantics [Adya, 1999; Cerone and Gotsman, 2016; Cerone et al., 2015a], it is easier to use our semantics to verify implementation protocols via trace refinement (in Chapter 6).

However, before we show the two applications, we have to justify our semantics, proving that our definitions of consistency models in our operational semantics are equivalent to well-known declarative definitions on abstract executions in Chapter 5: given an consistency model M an execution test $\mathtt{ET}_M$, we show that the set of all the reachable states (of the form of kv-stores) in our semantics, $\bigcup_{\mathrm{P}\in\mathrm{PROGS}}[\![\mathrm{P}]\!]_{\mathtt{ET}_M}$ defined in Def. 4.24, is equivalent to the set of reachable states (of the form of abstract executions) in the declarative semantics for M.

Chapter 5

Correctness of Execution Tests

We prove the correctness of execution tests with respect to axiomatic definitions on abstract executions. First, we demonstrate that there is a bijection between our kv-stores and dependency graphs in Section 5.1. In Section 5.2, we give an alternative operational semantics on abstract executions. Using the bijection between kv-stores and dependency graphs, and the connection between dependency graphs and abstract executions [Cerone et al., 2017; Cerone and Gotsman, 2016], in Section 5.3, we demonstrate connections between traces obtained by our semantics on kv-store and traces obtained by the semantics on abstract executions. In Section 5.4, we define *soundness* and *completeness constructors* for proving the correctness of our definitions of consistency models using execution tests. Using these two constructors, we only need to prove a notion of *soundness* and *completeness* between execution tests and the axioms on abstract executions, and the constructors lift the result to the level of traces. Finally, in Section 5.5, we show that definitions of consistency models using execution tests defined in Section 4.3 are sound and complete with respect to the definitions using axioms on abstract executions.

5.1 Correspondence to Dependency Graph

Dependency graphs [Adya, 1999] provide one well-known formalism for specifying consistency models. Adya [1999] originally used dependency graphs to model ANSI-SQL isolation levels, which are the first formal and abstract definitions, in contrast to previous definitions using reference implementations or anomalies.

A dependency graph $\mathcal{G}$ is a directed labelled graph, where the nodes denote transactions, and the edges denote *dependencies* between transactions, including session order SO, *write-read* dependency WR, *write-write* dependency WW and *read-write* anti-dependency RW.

Definition 5.1 (Dependency graph)**.** *Given the session order defined in Def. 4.2, the set of* dependency graphs*,* $\mathit{DGraphs} \ni \mathcal{G}$*, is defined by:*

$$\mathit{DGraphs} \stackrel{def}{=} \{(\mathcal{T}, \mathsf{SO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \mid \mathtt{WfDGraph}\,(\mathcal{T}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})\}\,.$$

where the well-formed condition for a dependency graph, `WfDGraph`*, is defined by:*

$$\texttt{WfDGraph}\,(\mathcal{T}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \overset{def}{\Leftrightarrow} \mathcal{T} \in \left(\left(\mathit{TxID}_0 \overset{fin}{\rightharpoonup} \mathit{Fp}\right) \uplus \{t_0 \mapsto \{(\mathtt{W}, k, v_0) \mid k \in \mathit{Key}\}\}\right) \wedge \forall k \in \mathit{Key}.\ \begin{pmatrix} \texttt{Wfwr}\,(\mathsf{WR}, \mathcal{T}, k) \wedge \texttt{Wfww}\,(\mathsf{WW}, \mathcal{T}, k) \\ \wedge\ \texttt{Wfrw}\,(\mathsf{WR}, \mathsf{WW}, \mathsf{RW}, k) \end{pmatrix}$$

and the well-formed conditions, `Wfwr`*,* `Wfww` *and* `Wfrw`*, for the three dependency relations are defined by:*

$$\texttt{Wfwr}\,(\mathsf{WR}, \mathcal{T}, k) \overset{def}{\Leftrightarrow} \tag{5.1}$$

$$\forall t, t', t''.\,(t, t') \in \mathsf{WR} \Rightarrow (\mathtt{W}, k, \_) \in \mathcal{T}(t) \wedge (\mathtt{R}, k, \_) \in \mathcal{T}(t') \tag{5.2}$$

$$(\mathtt{R}, k, \_) \in \mathcal{T}(t') \wedge \exists t^*.\,(\mathtt{W}, k, \_) \in \mathcal{T}(t^*) \Rightarrow (t^*, t') \in \mathsf{WR} \tag{5.3}$$

$$(t, t') \in \mathsf{WR} \Rightarrow (t', t) \notin \mathsf{SO}^? \tag{5.4}$$

$$(t', t) \in \mathsf{WR} \wedge (t'', t) \in \mathsf{WR}\,(\mathcal{T}, k) \Rightarrow t' = t'' \tag{5.5}$$

$$\texttt{Wfww}\,(\mathsf{WW}, \mathcal{T}, k) \overset{def}{\Leftrightarrow} \tag{5.6}$$

$$\forall t, t'.\,(\mathtt{W}, k, \_) \in \mathcal{T}(t) \wedge (\mathtt{W}, k, \_) \in \mathcal{T}(t') \Rightarrow t = t' \vee (t, t') \in \mathsf{WW} \vee (t', t) \in \mathsf{WW} \quad \text{(total-WW)}$$

$$(t, t') \in \mathsf{WW} \Rightarrow (\mathtt{W}, k, \_) \in \mathcal{T}(t) \wedge (\mathtt{W}, k, \_) \in \mathcal{T}(t') \tag{5.7}$$

$$(t, t') \in \mathsf{WW} \Rightarrow t' \neq t \wedge (t', t) \notin \mathsf{WW} \quad \text{(irref-asym-WW)}$$

$$\mathsf{WW} = \mathsf{WW}^+ \quad \text{(transitive-WW)}$$

$$(t, t') \in \mathsf{WW} \Rightarrow t' \neq t_0 \tag{5.8}$$

$$(t, t') \in \mathsf{WW} \Rightarrow (t', t) \notin \mathsf{SO} \tag{5.9}$$

$$\texttt{Wfrw}\,(\mathsf{WR}, \mathsf{WW}, \mathsf{RW}, k) \overset{def}{\Leftrightarrow} \tag{5.10}$$

$$\mathsf{RW} = \left\{(t, t') \;\middle|\; \exists t''.\,(t'', t) \in \mathsf{WR} \wedge (t'', t') \in \mathsf{WW} \wedge t \neq t'\right\}. \tag{5.11}$$

Given a dependency graph $\mathcal{G}$*, let* $\mathcal{T}_\mathcal{G}, \mathsf{WR}_\mathcal{G}, \mathsf{WW}_\mathcal{G}$ *and* $\mathsf{RW}_\mathcal{G}$ *return the first, the third to fifth projection respectively, and* $\mathsf{WR}_\mathcal{G}(k), \mathsf{WW}_\mathcal{G}(k)$ *and* $\mathsf{RW}_\mathcal{G}(k)$ *be the relations on the key* k*. Let notations* $t \in \mathcal{G}$ *and* $o \in \mathcal{G}(t)$ *denote* $t \in \mathsf{Dom}(\mathcal{T})$ *and* $o \in \mathcal{T}(t)$ *respectively.*

Fig. 5.1 gives an example dependency graph. Each node in a dependency graph is a fingerprint labelled with the unique transaction identifier. The special initialisation transaction t_0 initialises all keys, with a set of *infinite* write operations. Recall that there are infinite many of keys. Apart from the initialisation transaction that contains infinite write operations, other transactions must contain finite operations. Edges are labelled with dependency relations $\mathsf{SO}, \mathsf{WR}, \mathsf{WW}$ and RW. Session order SO is defined in Def. 4.2. The write-read dependency, $(t, t') \in \mathsf{WR}(k)$, means that a transaction t' read from a value of k written by another transaction t (Eq. (5.2)). Note that any read operation must read from a transaction (Eqs. (5.3) and (5.5)). The write-read dependency must agree with session order (Eq. (5.4)). Given a key k, the write-write

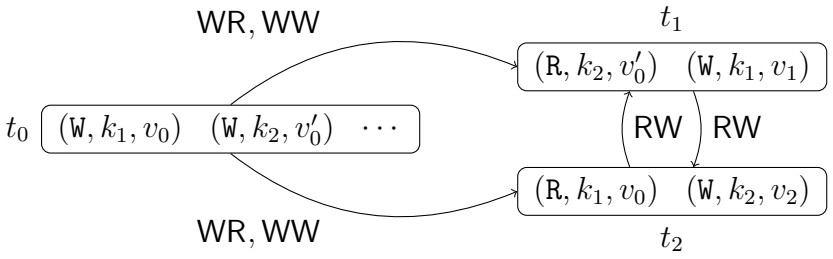


Figure 5.1: Dependency graph

dependency on k is a strict total order, that is, a total, irreflexive asymmetric and transitive relation over set of transactions $S \stackrel{def}{=} \{t \mid (\mathtt{W}, k, \_) \in \mathcal{T}(t)\}$:

(1) Eq. (total-WW) means WW is a total relation over the set S;

(2) Eq. (5.7) means that WW does not contain transactions that are not in S;

(3) Eq. (irref-asym-WW) means WW is irreflexive and asymmetric;

(4) Eq. (transitive-WW) means WW is transitive.

Given two transactions t, t', if $(t, t') \in \mathsf{WW}(k)$, then intuitively t' overwrites a value of k previously written by t. The initial transaction t_0 writes the initial value of k, hence $(t_0, t) \in \mathsf{WW}(k)$ for any transaction t that writes k in Eq. (5.8). The write-write dependency must agree on session order in Eq. (5.9). Last, read-write anti-dependency RW is derived from WR and WW. Intuitively, if $(t, t') \in \mathsf{RW}(k)$, the transaction t read a value v on the key k written by some transaction t'' that is later overwritten by t'.

In dependency graphs, consistency models are specified using axioms that rule out invalid graphs that contain certain cycles. For example, snapshot isolation require that graphs only contain cycles with at least *two adjacent read-write anti-dependency edges*, that is, relation $(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{G}} \cup \mathsf{WW}_{\mathcal{G}}); \mathsf{RW}_{\mathcal{G}}^{-1}$ is acyclic [Adya, 1999; Cerone and Gotsman, 2016].

In our kv-store semantics, we adopt the same names for the dependency relations between transactions. This is to emphasise the similarity between the dependency relations on kv-stores and the dependency relations on dependency graphs. In fact, there is a bijection between our global kv-stores and dependency graphs. We first show how to extract a dependency graph from a kv-store.

Definition 5.2 (Kv-stores to dependency graphs)**.** *Given a kv-store $\mathcal{K} \in KVS$, the dependency graph induced by the kv-store, written* $\mathsf{KToD}(\mathcal{K})$*, is defined by:*

$$\mathsf{KToD}(\mathcal{K}) \stackrel{def}{=} (\mathsf{KToT}(\mathcal{K}), \mathsf{SO}, \mathsf{WR}_{\mathcal{K}}, \mathsf{WW}_{\mathcal{K}}, \mathsf{RW}_{\mathcal{K}})$$

where $\mathsf{KToT}(\mathcal{K})$ *is defined by*

$$\mathsf{KToT}(\mathcal{K}) = \lambda t \in \mathcal{K}.\ \{(\mathtt{W}, k, v) \mid \exists i \in \mathbb{N}.\, \mathcal{K}(k,i) = (v, t, \_)\} \uplus \{(\mathtt{R}, k, v) \mid \exists i \in \mathbb{N}.\, \exists T \subseteq \mathit{TxID}.\, \mathcal{K}(k,i) = (v, \_, T) \wedge t \in T\}.$$

Given a kv-store $\mathcal{K}$, the dependency graph induced by the kv-store is given by $\mathsf{KToD}(\mathcal{K})$. The auxiliary function $\mathsf{KToT}(\mathcal{K})$ converts the kv-store $\mathcal{K}$ to the set of nodes in the dependency graph. The three dependency relations on the dependency graph are exactly the same dependency relations on kv-store defined in Def. 4.33. It is easy to see that the dependency graph induced by the kv-store $\mathsf{KToD}(\mathcal{K})$ is well-formed. More detail is given in Prop. A.4 on page 189.

Next, we show how to convert a dependency graph $\mathcal{G}$ to a kv-store via function DToK.

Definition 5.3 (Dependency graphs to kv-stores)**.** *Given a dependency graph* $\mathcal{G}$*, and a transaction* t *that writes a key* k*, the* version *for* k *written by* t *is defined by:*

$$\mathsf{VerOf}(\mathcal{G}, k, t) \stackrel{def}{=} \mathtt{let}\ \ (\mathtt{W}, v, k) \in \mathcal{G}(t)\ \mathtt{in}\ \ (v, t, \{t' \mid (t, t') \in \mathsf{WR}_{\mathcal{G}}(k)\}).$$

The version list *for* k*, written* $\mathsf{VerListOf}(\mathcal{G}, k)$*, is defined by:*

$$\mathsf{VerListOf}(\mathcal{G}, k) \stackrel{def}{=} \mathtt{let}\ T = \{t, t' \mid (t, t') \in \mathsf{WW}_{\mathcal{G}}(k)\}\ \mathtt{in}\ [\mathsf{VerOf}(\mathcal{G}, k, \mathsf{WWnth}(\mathcal{G}, T, 0)), \cdots, \mathsf{VerOf}(\mathcal{G}, k, \mathsf{WWnth}(\mathcal{G}, T, |T| - 1))]$$

$$\mathsf{WWnth}(\mathcal{G}, T, 0) \stackrel{def}{=} \mathsf{FirstWW}(\mathsf{WW}_{\mathcal{G}})$$

$$\mathsf{WWnth}(\mathcal{G}, T, n+1) \stackrel{def}{=} \mathsf{WWnth}(\mathcal{G}, T \setminus \{\mathsf{FirstWW}(\mathsf{WW}_{\mathcal{G}})\}, n)$$

$\mathsf{FirstWW}(\mathsf{WW}_{\mathcal{G}})t$ *where* $\forall t' \in T.\, t = t' \vee (t, t') \in \mathsf{WW}_{\mathcal{G}}(k)$

The kv-store induced by $\mathcal{G}$*, written* $\mathsf{DToK}(\mathcal{G})$*, is defined by:*

$$\mathsf{DToK}(\mathcal{G}) = \lambda k \in \mathit{Key}.\, \mathsf{VerListOf}(\mathcal{G}, k).$$

Given a key k and a transaction t that writes k, the version $\mathsf{VerOf}(\mathcal{G}, k, t)$ comprises the value of k written by t, the writer being t, and the reader set that contains all transactions t' that read from t, that is, $(t, t') \in \mathsf{WR}$. Given all versions of k, function $\mathsf{VerListOf}(\mathcal{G}, k)$ determines the order over the versions with respect to $\mathsf{WW}(k)$ relation. Recall that if $(t, t') \in \mathsf{WW}_{\mathcal{G}}(k)$, then t' committed after t. This means that the version written by t' precedes the version written by t in the list of $\mathsf{VerListOf}(\mathcal{G}, k)$. It is straightforward that $\mathsf{DToK}(\mathcal{G})$ satisfies the well-formed conditions for kv-store in Def. 4.5. More detail is given in Prop. A.5 on page 191.

We now show the bijection between our kv-stores and dependency graphs. This is a key intermediate step to link our global kv-stores to abstract executions.

Theorem 5.4 (Bijection between kv-stores and dependency graphs)**.** *There is a bijection between kv-stores and dependency graphs.*

Proof. Because $\mathsf{KToD}(\mathcal{K})$ and $\mathsf{DToK}(\mathcal{G})$ are well-defined respectively (detail is given in Props. A.4 and A.5), it remains to prove $\mathcal{K} = \mathsf{DToK}(\mathsf{KToD}(\mathcal{K}))$ and $\mathcal{G} = \mathsf{KToD}(\mathsf{DToK}(\mathcal{G}))$.

(1) [**Case**: $\mathcal{K} = \mathsf{DToK}(\mathsf{KToD}(\mathcal{K}))$] Let $\mathcal{G} = \mathsf{KToD}(\mathcal{K})$. Fix a key k and

$$\mathcal{K}(k) = [(v_0, t_0, T_0), \cdots, (v_n, t_n, T_n)].$$

By the definition of KToD, it follows that: (i) $(\mathtt{W}, k, v_i) \in \mathcal{G}(t_i)$; (ii) for all $t \in T_i$, $(\mathtt{R}, k, v_i) \in \mathcal{G}(t)$; and (iii) $\mathsf{WW}_{\mathcal{G}} = \mathsf{WW}_{\mathcal{K}}$ and $\mathsf{WR}_{\mathcal{G}} = \mathsf{WR}_{\mathcal{K}}$. Then by definition of DToK, specifically $\mathsf{VerListOf}$, it is easy to see $\mathsf{DToK}(\mathcal{G})(k) = \mathsf{VerListOf}(\mathcal{G}, k) = \mathcal{K}(k)$.

(2) [**Case**: $\mathcal{G} = \mathsf{KToD}(\mathsf{DToK}(\mathcal{G}))$] Let $\mathcal{K} = \mathsf{DToK}(\mathcal{G})$. First, we prove $\mathcal{T}_{\mathcal{G}} = \mathcal{T}_{\mathsf{KToD}(\mathcal{K})}$. Consider a transaction t and an operation o such that $o \in \mathcal{G}(t)$.

(i) [**Case**: $o = (\mathtt{R}, k, v)$] Because $\mathcal{G}$ is well-formed, there exists a transaction t' such that $(t', t) \in \mathsf{WR}_{\mathcal{G}}(k)$. By definition of DToK, there exists an index i and a reader set T such that $o \in \mathcal{G}(t) \Leftrightarrow \mathcal{K}(k, i) = (v, t', T) \wedge t \in T$. Then by definition of KToD, we have $o \in \mathsf{KToD}(\mathcal{K})(t)$. This means that $o \in \mathcal{G}(t) \Leftrightarrow o \in \mathsf{KToD}(\mathcal{K})(t)$.

(ii) [**Case**: $o = (\mathtt{W}, k, v)$] Because $\mathcal{G}$ is well-formed, that is, $\mathsf{WW}_{\mathcal{G}}(k)$ is a total order on all transactions that wrote *key*; by definition of DToK, there exists an index i and a reader set T such that $o \in \mathcal{G}(t) \Leftrightarrow \mathcal{K}(k, i) = (v, t', T)$. By definition of KToD, we have $o \in \mathsf{KToD}(\mathcal{K})(t)$. This means that $o \in \mathcal{G}(t) \Leftrightarrow o \in \mathsf{KToD}(\mathcal{K})(t)$.

Second, we prove $\mathsf{WR}_{\mathcal{G}} = \mathsf{WR}_{\mathsf{KToD}(\mathcal{K})}$ and $\mathsf{WW}_{\mathcal{G}} = \mathsf{WW}_{\mathsf{KToD}(\mathcal{K})}$. consider a key k and the write-write relation on all transactions $t_0, \cdots, t_n$ that write k, $t_0 \xrightarrow{\mathsf{WW}_{\mathcal{G}}(k)} \cdots \xrightarrow{\mathsf{WW}_{\mathcal{G}}(k)} t_n$. Consider T_i for $0 \leq i < n$ such that $\forall t \in T_i.\, (t_i, t) \in \mathsf{WR}_{\mathcal{G}}$. By the definition of DToK, $\mathcal{K}(k) = \mathsf{DToK}(\mathcal{G})(k) = [(v_0, t_0, T_0), \cdots, (v_n, t_n, T_n)]$. Now by the definition of $\mathsf{WW}_{\mathcal{K}}$ and $\mathsf{WR}_{\mathcal{K}}$, we have $\mathsf{WR}_{\mathcal{G}}(k) =\in \mathsf{WR}_{\mathcal{K}}(k)$ and $\mathsf{WW}_{\mathcal{G}}(k) = \mathsf{WW}_{\mathcal{K}}(k)$. By definition of KToD, then we have $\mathsf{WW}_{\mathcal{K}} = \mathsf{WW}_{\mathsf{KToD}(\mathcal{K})}$ and $\mathsf{WR}_{\mathcal{K}} = \mathsf{WR}_{\mathsf{KToD}(\mathcal{K})}$, it follows $\mathsf{WW}_{\mathcal{G}} = \mathsf{WW}_{\mathsf{KToD}(\mathsf{DToK}(\mathcal{G}))}$ and $\mathsf{WR}_{\mathcal{G}} = \mathsf{WR}_{\mathsf{KToD}(\mathsf{DToK}(\mathcal{G}))}$. Last, because $\mathsf{RW}_{\mathcal{G}}$ relation can be derived from $\mathsf{WW}_{\mathcal{G}}$ and $\mathsf{WR}_{\mathcal{G}}$, it is easy to see $\mathsf{RW}_{\mathcal{G}} = \mathsf{RW}_{\mathsf{KToD}(\mathcal{K})}$. ■

Consistency models in dependency graphs are specified as checks of acyclicicity. For example, snapshot isolation requires the relation $\left((\mathsf{WR}_{\mathcal{G}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{G}}) ; \mathsf{RW}_{\mathcal{G}}^{?}\right)^{+}$ be acyclic [Adya, 1999]. In this thesis, we prove that our definitions using execution tests in Section 4.3 are equivalent to definitions on abstract executions in the following chapter. We only use dependency graphs as an intermediate step to link the individual kv-store to abstract execution.

Hence, we refer the readers to [Cerone et al., 2015a,b, 2017; Cerone and Gotsman, 2016] for the definitions of other well-known consistency models on dependency graphs. Note that dependency graphs can be used to formally define consistency models weaker than read atomic, such as read commit and read uncommit [Adya, 1999], however, it is out of the scope of this thesis.

5.2 Operational Semantics on Abstract Execution

Abstract executions [Burckhardt et al., 2012; Cerone et al., 2015a] are an alternative declarative formalism for defining consistency models. We give the formal definition of abstract executions adopted from [Cerone et al., 2015a]. We then propose an alternative operational semantics on abstract executions in Section 5.2, which is used in Section 5.3 and Section 5.5 to prove correctness of our definitions of consistency models with respect to the axiomatic definitions on abstract executions. This alternative semantics is parametrised by an axiom $\mathcal{A}$, which gives rise to a consistency model. The goal is to prove the set of kv-store induced by an execution test is equivalent to the set of abstract executions induced by an axiom. Given $\mathcal{A}$, we show that the set of abstract executions induced by the alternative semantics is the same set induced by applying the axiom directly to all possible graphs (Section 5.2). We show that the set of kv-store program traces under $\texttt{ET}_\top$ is equivalent to the set of abstract executions test traces with the axiom being the maximum (Section 5.3). We propose soundness and complete constructors, in the sense that if an execution test $\texttt{ET}$ and an axiom $\mathcal{A}$ satisfy certain conditions, these two constructors lift them to the level of traces. Last, we use these two constructors to show that our definitions of consistency models are correct with respect to the definitions on abstract executions.

5.2.1 Declarative Model: Abstract Executions

Abstract executions [Burckhardt et al., 2012; Cerone et al., 2015a] are an alternative formalism for defining consistency models. As with dependency graphs, an abstract execution $\mathcal{X}$ is a directed graph where nodes represent transactions, and edges represent certain relations between transactions:

(1) *visibility relation* which means that, if $(t, t') \in \mathsf{VIS}$, then transaction t' *observes* t, or in other words, t is *visible* to t; and

(2) *arbitration relation*, also known as *arbitration order*, which means that if $(t, t') \in \mathsf{AR}$, then the update of the t' *overwrite* the update of t, or in other words, t *happens before* t'.

An example abstract execution graph is given in Fig. 5.2. There are three transactions t_0, t_1, t_2 where t_0 is the initialisation transaction that initialises all keys to value v_0. Transactions in

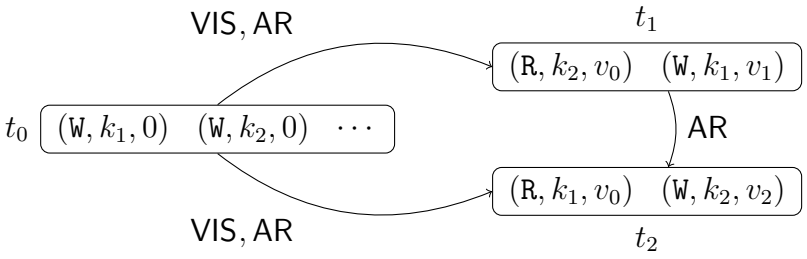


Figure 5.2: Abstract execution

abstract executions are totally ordered by arbitration relation AR, for example, $t_0 \xrightarrow{\mathsf{AR}} t_1 \xrightarrow{\mathsf{AR}} t_2$. Transaction t_1 reads key k_2 with value v_0 and update key k_1 to v_1. Then transaction t_2, that only observe the effect of t_0 given by the visibility relation VIS, reads key k_1 with value v_0 and update key k_2 to v_2. This mean that k_1 and k_2 have values v_1 and v_2 respectively, in the final state of the database.

Definition 5.5 (Abstract executions)**.** *Given the definition of* SO *(Def. 4.2), the set of* abstract executions*, $AExecTS \ni \mathcal{X}$, is defined by: $AExecTS \stackrel{def}{=} \{(\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{AR}) \mid \mathtt{WfAExec}\,(\mathcal{T}, \mathsf{VIS}, \mathsf{AR})\}$ where the well-formed condition,* `WfAExec`*, is defined by:*

$$\begin{aligned}&\mathtt{WfAExec}\,(\mathcal{T}, \mathsf{VIS}, \mathsf{AR}) \stackrel{def}{\Leftrightarrow}\\ &\quad \mathcal{T} \in \left(\left(TxID_0 \stackrel{fin}{\rightharpoonup} Fp\right) \uplus \{t_0 \mapsto \{(\mathtt{W}, k, v_0) \mid k \in Key \wedge v_0 \in \mathsf{InitialValue}\,(k)\}\}\right)\\ &\qquad\qquad \mathtt{Wfar}\,(\mathsf{AR}, \mathcal{T}) \wedge \mathtt{Wfvis}\,(\mathsf{VIS}, \mathcal{T}, \mathsf{AR})\,.\end{aligned}$$

The well-formed conditions on visibility and arbitration relations, `Wfar` *and* `Wfvis`*, are defined:*

$$\begin{aligned}&\mathtt{Wfar}\,(\mathsf{AR}, \mathcal{T}) \stackrel{def}{\Leftrightarrow} \forall t, t' \in \mathsf{Dom}\,(\mathcal{T})\,.\\ &\qquad t = t' \vee (t, t') \in \mathsf{AR} \vee (t', t) \in \mathsf{AR} && \text{(total-AR)}\\ &\qquad \wedge\, (t, t) \notin \mathsf{AR} && \text{(irreflexive-AR)}\\ &\qquad \wedge\, (t, t') \in \mathsf{AR} \Rightarrow (t', t) \notin \mathsf{AR} && \text{(asymmetric-AR)}\\ &\qquad \wedge\, \mathsf{AR} = \mathsf{AR}^+ && \text{(transitive-AR)}\\ &\qquad \wedge\, (t_0, t) \in \mathsf{AR} && (5.12)\\ &\qquad \wedge\, \mathsf{SO} \subseteq \mathsf{AR} && (5.13)\\ &\mathtt{Wfvis}\,(\mathsf{VIS}, \mathcal{T}, \mathsf{AR}) \stackrel{def}{\Leftrightarrow} \forall t, t' \in \mathsf{Dom}\,(\mathcal{T})\,.\, \forall k \in Key.\, \forall v \in Value.\\ &\qquad ((t, t') \in \mathsf{VIS} \Rightarrow (t', t) \notin \mathsf{SO}) && (5.14)\\ &\qquad \wedge\, \mathsf{VIS} \subseteq \mathsf{AR} && (5.15)\\ &\qquad \wedge\, (t_0, t) \in \mathsf{VIS} && (5.16)\\ &\qquad \wedge \begin{pmatrix}(\mathtt{R}, k, v) \in \mathcal{T}(t)\\ \Rightarrow (\mathtt{W}, k, v) \in \mathcal{T}\big(\mathsf{MaxVisTrans}\,\big((\mathcal{T}, \mathsf{VIS}, \mathsf{AR}), \mathsf{VIS}^{-1}(t), k\big)\big)\end{pmatrix} && (5.17)\end{aligned}$$

where $\mathsf{MaxVisTrans}\left((\mathcal{T}, \mathsf{VIS}, \mathsf{AR}), T, k\right)$ *defined by:*

$$\mathsf{MaxVisTrans}\left((\mathcal{T}, \mathsf{VIS}, \mathsf{AR}), T, k\right) \stackrel{def}{=} \mathsf{Max}_{\mathsf{AR}}\left(\{t \mid \exists v \in \textsc{Value}.\, t \in T \wedge (\mathtt{W}, k, v) \in \mathcal{T}(t)\}\right)$$

with $\mathsf{Max}_{\mathsf{AR}}\left(T'\right)$ *returning the maximum transaction in* T' *with respect to* AR*. Given an abstract execution* $\mathcal{X}$*, let* $\mathcal{T}_{\mathcal{X}}$*,* $\mathsf{VIS}_{\mathcal{X}}$ *and* $\mathsf{AR}_{\mathcal{G}}$ *be the first, third and fourth projections. Notation* $t \in \mathcal{G}$ *and* $\mathcal{G}(t)$ *denote* $t \in \mathsf{Dom}\left(\mathcal{T}\right)$ *and* $\mathcal{T}(t)$ *respectively. Let the initial abstract execution* $\mathcal{X}_0 = \left(\{t_0 \mapsto \{(\mathtt{W}, k, v_0) \mid k \in \textsc{Key} \wedge v_0 \in \mathsf{InitialValue}\left(k\right)\}\right\}, \emptyset, \emptyset\right)$*.*

For a well-formed abstract execution, the arbitration relation AR is a strict total order over all the transactions (Eq. (total-AR), Eq. (asymmetric-AR), Eq. (irreflexive-AR) and Eq. (transitive-AR)) starting from the initialisation transaction (Eq. (5.12)), and agrees with the session order SO (Eq. (5.13)). The visibility relation VIS agrees with the session order (Eq. (5.14)) and arbitrary order (Eq. (5.15)) Transactions always see the initialisation transaction t_0 (Eq. (5.16)). Lastly, we only consider abstract executions that apply the *last-write-wins* policy, that is, a transaction reading k always fetches the latest visible write (VIS-predecessor) on k (Eq. (5.17)).

Given an abstract execution $\mathcal{X}$, A set of *visibility axioms* $\mathcal{A}$, defined in Def. 5.6, specifies the *minimum visibility relation* in $\mathcal{X}$: that is, for all visibility axiom $A \in \mathcal{A}$, the visibility relation $\mathsf{VIS}_{\mathcal{X}}$ satisfies $A(\mathcal{X}) \subseteq \mathsf{VIS}_{\mathcal{X}}$. Each $\mathcal{A}$ gives rise to a consistency model defined in Def. 5.7. Def. 5.7 is a well-known definition that we adopt from [Cerone et al., 2015a,b, 2017; Cerone and Gotsman, 2016]. In this thesis, we enforce a well-form condition on the set of visibility axioms $\mathcal{A}$, which is satisfied by the definitions of consistency models presented in Fig. 4.6a. Given two abstract executions $\mathcal{X}, \mathcal{X}'$, if they contain the same sub-graph of transactions T, if and only if, the sub-graphs of $\mathcal{X}$ and $\mathcal{X}'$ by projecting to T are the same. The well-formed condition for $\mathcal{A}$ requires that for any sub-graphs of transactions T between two arbitrary abstract executions $\mathcal{X}, \mathcal{X}'$, the subsets of minimum visibility edges by projecting to the sub-graph T in both abstract executions, that is, $A(\mathcal{X}) \cap (T \times T)$ and $A(\mathcal{X}') \cap (T \times T)$ respectively, are the same. This intuitively means that $\mathcal{A}$ must be *local* to T, other transactions and edges in $\mathcal{X}$ and/or $\mathcal{X}'$ cannot affect the minimum edges for T.

Definition 5.6 (Visibility axioms)**.** *The set of* visibility relation axioms *or* axioms*,* $\textsc{VisAxioms} \ni A$*, is defined by:* $\textsc{VisAxioms} \stackrel{def}{=} \textsc{AExecTs} \to \mathcal{P}\left(\textsc{TxID} \times \textsc{TxID}\right)$*, such that whenever two abstract executions* $\mathcal{X}, \mathcal{X}'$ *contain the same subset of transactions* T*, written* $\mathcal{X} \simeq_T \mathcal{X}'$*, then* $A(\mathcal{X}) \cap (T \times T) = A(\mathcal{X}') \cap (T \times T)$*. Two abstract executions* $\mathcal{X}, \mathcal{X}'$ contain *the same sub-graph of transactions* T*, if and only if*

$$\forall t, t' \in T.\, \mathcal{T}_{\mathcal{X}}\left(t\right) = \mathcal{T}_{\mathcal{X}}\left(t'\right) \wedge \left(t \xrightarrow{\mathsf{VIS}_{\mathcal{X}}} t' \Leftrightarrow t \xrightarrow{\mathsf{VIS}_{\mathcal{X}'}} t'\right) \wedge \left(t \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t' \Leftrightarrow t \xrightarrow{\mathsf{AR}_{\mathcal{X}'}} t'\right).$$

The subsets of $\textsc{VisAxioms}$ *are ranged over* $\mathcal{A}, \mathcal{A}_0, \mathcal{A}', \cdots$*.*

Definition 5.7 (Consistent models on abstract executions)**.** *An abstract execution* $\mathcal{X}$ *satisfies a set of axioms* $\mathcal{A}$*, written* $\mathtt{AExecSat}\left(\mathcal{X}, \mathcal{A}\right)$*, if and only if* $\forall A \in \mathcal{A}.\, A(\mathcal{X}) \subseteq \mathsf{VIS}_{\mathcal{X}}$*. The set of*

abstract executions induced by a set of axioms $\mathcal{A}$ is defined by:

$$\mathsf{ConsisModelAxioms}\,(\mathcal{A}) \stackrel{def}{=} \{\mathcal{X} \mid \mathtt{AExecSat}\,(\mathcal{X}, \mathcal{A})\}\,.$$

5.2.2 Operational Semantics on Abstract Execution

We introduce an operational semantics on abstract executions. This operational semantics is an intermediate step for proving the correctness of execution tests defined in Fig. 4.6a. Given a consistency model M, In this section, we prove that reachable abstract executions in the operational semantics satisfied the declarative definition of M, and vice versa. Then, in Sections 5.3 to 5.5, we show that trace simulation between operational semantics on abstract executions for M and operational semantics on kv-stores for M, for all M presented in Fig. 4.6a.

The operational semantics on abstract executions is given in Fig. 5.3. Each labelled transition is of the form $(\mathcal{X}, \mathcal{E})\,, \mathtt{P} \xrightarrow{\iota}_{\mathcal{A}} (\mathcal{X}', \mathcal{E}')\,, \mathtt{P}'$. The label is either a primitive command, $(cl, \bullet)$, or a transaction commit step, $(cl, T, \mathcal{F})$, with the given visible set of transactions T. The semantics is parametrised by the set of visibility axioms $\mathcal{A}$.

Definition 5.8 (Abstract execution labels)**.** *The set of abstract execution labels, $\textsc{ALabels} \ni \iota$, is defined by*

$$\textsc{ALabels} \stackrel{def}{=} \{(cl, \bullet) \mid cl \in \textsc{CID}\} \cup \{(cl, T, \mathcal{F}) \mid cl \in \textsc{CID} \wedge T \subseteq \textsc{TxID} \wedge \mathcal{F} \in \textsc{Fp}\}\,.$$

In Fig. 5.3, except the rule AAtomicTrans, other rules are standard and mimic the counterparts in Figs. 4.2 and 4.4. The rule AAtomicTrans describes how a client cl with the set of visible transactions T, commits a transaction $[\ \mathtt{T}\ ]$, in a similar way as the rule CAtomicTrans in Fig. 4.2. Prior to executing the transactional command $\mathtt{T}$, the client cl picks the set of visible transactions T (the second and third premises $t_0 \in T \wedge T \subseteq \mathsf{Dom}\,(\mathcal{T})$). This set of transactions determines an initial snapshot that contains the latest visible value of each key, captured by the fourth premise $\sigma \in \mathsf{AExecSnapshot}\,(\mathcal{X}, T)$ in AAtomicTrans. The function $\mathsf{AExecSnapshot}$ models the *last-write-win* policy. Given a set of visible transactions T, the snapshot induced by T contains the last value for each key. For example, in Fig. 5.4, if $T = \{t_0, t_1, t_2\}$, the snapshot $\mathsf{AExecSnapshot}\,(\mathcal{X}, T) = \{k_1 \mapsto v_1, k_2 \mapsto v_2\}$, because $t_0 \xrightarrow{\mathsf{AR}} t_1 \xrightarrow{\mathsf{AR}} t_2$. It is straightforward that $\mathsf{AExecSnapshot}$ is a valid snapshot, since $t_0 \in T$ and t_0 initialise all the keys in a well-formed abstract execution.

Definition 5.9 (Snapshots on abstract executions)**.** *The function,* $\mathsf{AExecSnapshot} : \textsc{AExecTs} \times \mathcal{P}\,(\textsc{TxID}) \to \textsc{Snapshot}$*, is defined by: for all keys $k \in \textsc{Key}$,*

$$\mathsf{AExecSnapshot}\,(\mathcal{X}, T) \stackrel{def}{=} \lambda k.\, \mathtt{let}\ t = \mathsf{MaxVisTrans}\,(\mathcal{X}, T, k)\ \mathtt{and}\ (\mathtt{W}, k, v) \in \mathcal{T}_{\mathcal{X}}\,(t)\ \ \mathtt{in}\ v$$

where $\mathsf{MaxVisTrans}$ *is defined in Def. 5.5.*

$\rightarrow_{\mathcal{A}} : (\textsc{AExects} \times \textsc{Stack} \times \textsc{Commands}) \times \textsc{ALabels} \times (\textsc{AExects} \times \textsc{Stack} \times \textsc{Commands})$

AAtomicTrans

$$\frac{\begin{array}{c}\mathcal{X} = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{AR}) \\ t_0 \in T \qquad T \subseteq \mathsf{Dom}(\mathcal{T}) \qquad \sigma \in \mathsf{AExecSnapshot}(\mathcal{X}, T) \qquad (s, \sigma, \emptyset), \mathtt{T} \rightsquigarrow^* (s', \sigma', \mathcal{F}), \mathtt{skip} \\ t \in \mathsf{NextAExecTxID}(\mathcal{X}, cl) \qquad \mathcal{X}' = \mathsf{UpdateAExec}(\mathcal{X}, T, \mathcal{F}, t) \qquad \forall A \in \mathcal{A}.\ A^{-1}(\mathcal{X}')(t) \subseteq T\end{array}}{(\mathcal{X}, s), [\ \mathtt{T}\ ] \xrightarrow{(cl, T, \mathcal{F})}_{\mathcal{A}} (\mathcal{X}', s'), \mathtt{skip}}$$

APrimitive

$$\frac{s \overset{\mathtt{C}_p}{\rightsquigarrow} s'}{(\mathcal{X}, s), \mathtt{C}_p \xrightarrow{(cl, \bullet)}_{\mathcal{A}} (\mathcal{X}', s'), \mathtt{skip}}$$

AChoice

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

AIter

$$\frac{}{(\mathcal{X}, s), \mathtt{C}^* \xrightarrow{(cl, \bullet)}_{\mathcal{A}} (\mathcal{X}, s), \mathtt{skip} + (\mathtt{C}\,;\,\mathtt{C}^*)}$$

ASeqSkip

$$\frac{}{(\mathcal{X}, s), \mathtt{skip}\,;\,\mathtt{C} \xrightarrow{(cl, \bullet)}_{\mathcal{A}} (\mathcal{X}, s), \mathtt{C}}$$

ASeq

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

$\rightarrow_{\mathcal{A}} : (\textsc{AExects} \times \textsc{ClientEnvs} \times \textsc{Progs}) \times \textsc{ALabels} \times (\textsc{AExecs} \times \textsc{ClientEnvs} \times \textsc{Progs})$

AProg

$$\frac{s = \mathcal{E}(cl) \qquad \mathtt{C} = \mathtt{P}(cl) \qquad \iota = (cl, \cdots) \qquad (\mathcal{X}, s), \mathtt{C} \xrightarrow{\iota}_{\mathcal{A}} (\mathcal{X}, s'), \mathtt{C}'}{(\mathcal{X}, \mathcal{E}), \mathtt{P} \xrightarrow{\iota}_{\mathcal{A}} (\mathcal{X}', \mathcal{E}[cl \mapsto s']), \mathtt{P}[cl \mapsto \mathtt{C}']}$$

Figure 5.3: Operational semantics on abstract executions

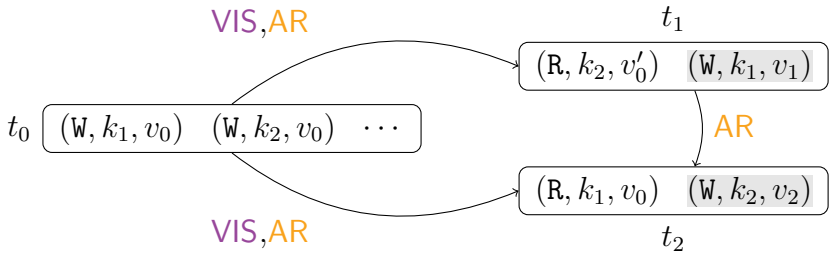


Figure 5.4: An example of $\mathsf{AExecSnapshot}(\mathcal{X}, T)$, where $T = \{t_0, t_1, t_2\}$

The transaction command T is executed with the initial snapshot σ, resulting a final fingerprint $\mathcal{F}$ where the final snapshot σ' is ignored (the fifth premise in AAtomicTrans). As with CAtomicTrans, the client cl is now ready to commit the transaction with the fingerprint $\mathcal{F}$:

(1) pick the next fresh transaction identifier $t \in \mathsf{NextAExecTxID}(\mathcal{K}, cl)$ (we allows any trans-

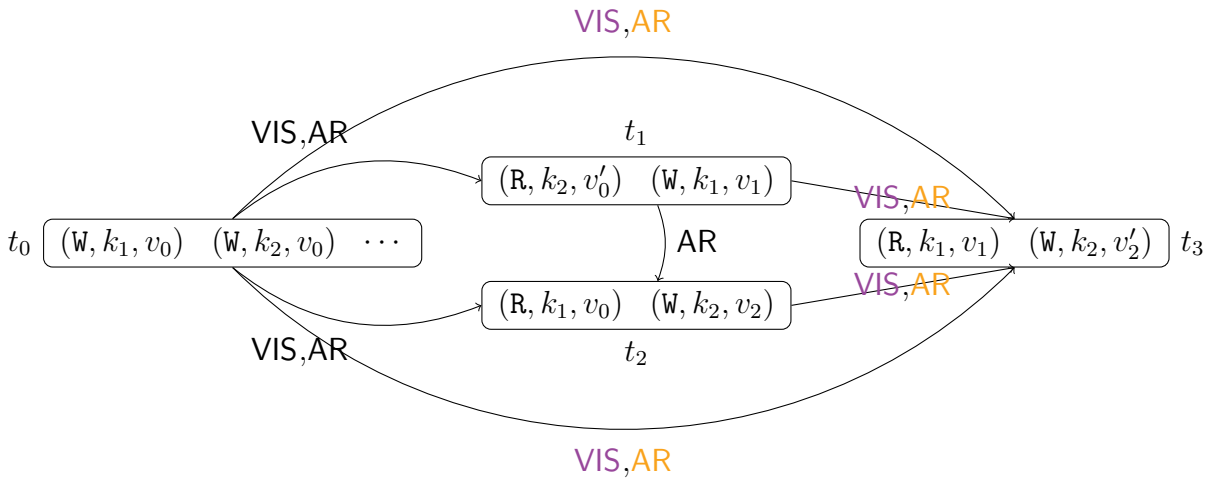


Figure 5.5: An example of $\mathcal{X}' = \mathsf{UpdateAExec}\,(\mathcal{X}, T, \mathcal{F}, t_3)$, where $T = \{t_0, t_1, t_2\}$ and $\mathcal{F} = \{(\mathtt{R}, k_1, v_1), (\mathtt{W}, k_2, v_2')\}$ and new edges are label with color, VIS in purple and AR in orange

action identifiers with bigger number, similar to Def. 4.19);

(2) compute the new abstract execution via $\mathcal{X}' = \mathsf{UpdateAExec}\,(\mathcal{X}, T, \mathcal{F}, t)$ (an example is given in Fig. 5.5); and

(3) check if the *new visibility edges* satisfy the axioms $\mathcal{A}$: that is, $A^{-1}(\mathcal{X}')(t) \subseteq T$ for all $A \in \mathcal{A}$.

Definition 5.10 (Fresh transaction identifiers and abstract execution update)**.** *Given an abstract execution $\mathcal{X}$ and a client cl, the set of* next available transaction identifiers*, written* $\mathsf{NextAExecTxID}\,(\mathcal{X}, cl)$*, is defined by:*

$$\mathsf{NextAExecTxID}\,(\mathcal{X}, cl) \stackrel{def}{=} \{t_{cl}^n \mid t_{cl}^n \in \mathit{TxID} \wedge \forall m \in \mathbb{N}.\, \forall t_{cl}^m \in \mathcal{X}.\, m < n\}\,.$$

Given a set of visible transactions $T \subseteq \mathit{TxID}$, a fingerprint $\mathcal{F} \in \mathit{Fp}$, and a fresh transaction $t \in \mathsf{NextAExecTxID}\,(\mathcal{X}, cl)$, the new abstract execution, written $\mathsf{UpdateAExec}\,(\mathcal{X}, T, \mathcal{F}, t)$*, is defined by:*

$$\mathsf{UpdateAExec}\,(\mathcal{X}, T, \mathcal{F}, t) \stackrel{def}{=} \mathtt{let}\ (\mathcal{T}, \mathsf{VIS}, \mathsf{AR}) = \mathcal{X}\ \mathtt{in}$$
$$(\mathcal{T} \cup \{t \mapsto \mathcal{F}\}, \mathsf{VIS} \cup \{(t', t) \mid t' \in T\}, \mathsf{AR} \cup \{(t', t) \mid t' \in \mathcal{X}\})\,.$$

Proposition 5.11 (Well-defined UpdateAExec function)**.** *Given an abstract execution $\mathcal{X} \in \mathit{AExecTs}$, a set of transactions $T \subseteq \mathcal{X}$ with $t_0 \in T$, a fresh transaction identifier $t \in \mathsf{NextAExecTxID}\,(\mathcal{K}, cl)$ for some client cl, and a fingerprint $\mathcal{F} \in \mathit{Fp}$ such that*

$$\forall k \in \mathit{Key}.\, \forall v \in \mathit{Value}.\, (\mathtt{R}, k, v) \in \mathcal{F} \Rightarrow \mathsf{MaxVisTrans}\,(\mathcal{X}, T, k)\,, \tag{5.18}$$

the new abstract execution $\mathsf{UpdateAExec}\,(\mathcal{X}, T, \mathcal{F}, t)$ *is a well-formed abstract execution.*

Proof sketch. This is straightforward since $t \in \mathsf{NextAExecTxID}\,(\mathcal{X}, cl)$ is a fresh transition identifier annotated with a greater number than any previous transaction identifiers in the same session. Therefore the AR of the new abstract execution is well-formed. The fingerprint contains the maximum visible value of each key k, if there is a read operation for k. Therefore the VIS of the new abstract execution is well-formed. The full proof is given in appendix A.3 on page 192. ■

This operational semantics on abstract executions are equally expressive as the axiomatic definitions in that, given a set of axioms $\mathcal{A}$, both formalisms yield the same set of abstract executions. We first define the notion of reachable abstract executions of a program $\mathtt{P}$, written $[\![\mathtt{P}]\!]_{\mathcal{A}}$.

Definition 5.12 (Abstract executions induced by programs)**.** *Given a program* $\mathtt{P}$*, the set of abstract executions obtained by executing the program under a set of axioms* $\mathcal{A}$ *is defined by:*

$$[\![\mathtt{P}]\!]_{\mathcal{A}} \stackrel{def}{=} \{\mathcal{X} \mid \exists n.\, (\mathcal{X}, \_, \_) = \mathsf{Last}\,(\mathsf{ATracesN}\,(\mathcal{A}, \mathtt{P}, n))\}$$

where $\mathsf{ATracesN}$ *is defined by:*

$$\begin{aligned}
\mathsf{ATracesN}\,(\mathcal{A}, \mathtt{P}, 0) &\stackrel{def}{=} \{(\mathcal{X}_0, \mathcal{E})\,, \mathtt{P} \mid \mathsf{Dom}\,(\mathtt{P}) \subseteq \mathsf{Dom}\,(\mathcal{E})\} \\
\mathsf{ATracesN}\,(\mathcal{A}, \mathtt{P}, n+1) &\stackrel{def}{=} \left\{\pi \xrightarrow{\iota}_{\mathcal{A}} (\mathcal{X}, \mathcal{E})\,, \mathtt{P}' \;\middle|\; \pi \in \mathsf{ATracesN}\,(\mathtt{P}, n) \wedge \mathsf{Dom}\,(\mathtt{P}') \subseteq \mathsf{Dom}\,(\mathcal{E})\right\}.
\end{aligned}$$

In Theorem 5.14, we prove that operational semantics on abstract executions is equivalent to the declarative semantics, that is $\bigcup_{\mathtt{P} \in \mathrm{PROGS}} [\![\mathtt{P}]\!]_{\mathcal{A}} = \mathsf{ConsisModelAxioms}\,(\mathcal{A})$ for a set of visibility axioms $\mathcal{A}$, where $\mathsf{ConsisModelAxioms}$ is the well-known definition defined in Def. 5.7.

We first introduce *cuts* of an abstract execution $\mathcal{X}$, which are used to construct traces where the final state $\mathcal{X}$. This means that for all transactions in $\mathcal{X}$, there is a trace,

$$\begin{aligned}
\mathcal{X}_0 \xrightarrow{\left(cl, t_1, \mathsf{VIS}^{-1}(t_1), \mathcal{T}_{\mathcal{X}}(t_1)\right)}_{\mathcal{A}} \cdots \xrightarrow{\left(cl, t_i, \mathsf{VIS}^{-1}(t_i), \mathcal{T}_{\mathcal{X}}(t_i)\right)}_{\mathcal{A}} \mathcal{X}_i \\
\xrightarrow{\left(cl, t_{i+1}, \mathsf{VIS}^{-1}(t_{i+1}), \mathcal{T}_{\mathcal{X}}(t_{i+1})\right)}_{\mathcal{A}} \cdots \xrightarrow{\left(cl, t_n, \mathsf{VIS}^{-1}(t_n), \mathcal{T}_{\mathcal{X}}(t_n)\right)}_{\mathcal{A}} \mathcal{X}_n
\end{aligned}$$

where $\mathcal{X}_n = \mathcal{X}$, and for each i, $\mathsf{VIS}^{-1}\,(t_i)$ is the set of visibility transactions and $\mathcal{T}_{\mathcal{X}}\,(t_i)$ is the fingerprint. We then define $\mathsf{AExectCut}\,(\mathcal{X}, i) \stackrel{def}{=} \mathcal{X}_i$. Note that the zero-cut is the graph that only contains the initialisation transaction.

Definition 5.13 (Cuts of abstract executions)**.** *The cut of an abstract execution* $\mathcal{X}$ *induced by the first* i *transactions, written* $\mathsf{AExectCut}\,(\mathcal{X}, i)$*, is defined by:*

$$\begin{aligned}
\mathsf{AExectCut}\,(\mathcal{X}, i) \stackrel{def}{=} \mathtt{let}\ T = \mathsf{ARClose}\,(\mathcal{X}, i)\ \mathtt{in} \\
(\lambda t \in T.\, \mathcal{T}_{\mathcal{X}}\,(t)\,, \mathsf{VIS}_{\mathcal{X}} \cap \{(t, t') \mid t, t' \in T\}\,, \mathsf{AR}_{\mathcal{X}} \cap \{(t, t') \mid t, t' \in T\})
\end{aligned}$$

where $\mathsf{ARClose}\,(\mathcal{X}, i)$ *is defined by:* $\mathsf{ARClose}\,(\mathcal{X}, i) \stackrel{def}{=} \{t_0, \cdots, t_i\}$ *for transactions* $t_0, \cdots, t_i$ *such that* $t_0 \xrightarrow{\mathsf{AR}_{\mathcal{X}}} \cdots \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t_i \xrightarrow{\mathsf{AR}_{\mathcal{X}}} \cdots \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t_{|\mathcal{X}|-1}$.

As explained before, given an abstract execution $\mathcal{X}$, it is easy to see that the i^{th} cut is the result of committing next transaction (with respect to AR) to the i^{th} cut. The detail of the proof is given in Prop. A.7 on page 194. In the trace induced by $\mathsf{AExectCut}$,

$$\mathsf{AExectCut}\,(\mathcal{X}, 0) \xrightarrow{\left(cl, t_1, \mathsf{VIS}^{-1}(t_1), \mathcal{T}_{\mathcal{X}}(t_1)\right)}_{\mathcal{A}} \cdots \xrightarrow{\left(cl, t_i, \mathsf{VIS}^{-1}(t_i), \mathcal{T}_{\mathcal{X}}(t_i)\right)}_{\mathcal{A}} \mathsf{AExectCut}\,(\mathcal{X}, i)$$
$$\xrightarrow{\left(cl, t_{i+1}, \mathsf{VIS}^{-1}(t_{i+1}), \mathcal{T}_{\mathcal{X}}(t_{i+1})\right)}_{\mathcal{A}} \cdots \xrightarrow{\left(cl, t_n, \mathsf{VIS}^{-1}(t_n), \mathcal{T}_{\mathcal{X}}(t_n)\right)}_{\mathcal{A}} \mathsf{AExectCut}\,(\mathcal{X}, n)$$

there is no program, yet it is easy to construct the program by the fingerprints, $\mathcal{T}_{\mathcal{X}}(t_1), \ldots, \mathcal{T}_{\mathcal{X}}(t_n)$. This is a key step to prove that an abstract execution induced by declarative semantics is reachable in the operational semantics.

Theorem 5.14 (Equal expressibility between declarative and operational semantics on abstract executions)**.** *For any* $\mathcal{A} \subseteq$ *VisAxioms, the operational semantics capture the same set of abstract executions as direct axiomatic definitions on abstract definitions, that is,* $\bigcup_{\mathtt{P} \in PROGS} [\![\mathtt{P}]\!]_{\mathcal{A}} = \mathsf{ConsisModelAxioms}\,(\mathcal{A})$.

Proof sketch. It is easy to see that $\bigcup_{\mathtt{P} \in \mathrm{PROGS}} [\![\mathtt{P}]\!]_{\mathcal{A}} \subseteq \mathsf{ConsisModelAxioms}\,(\mathcal{A})$ since each step in the traces from the program $\mathtt{P}$ is constrained by $\mathcal{A}$. The opposite way, $\bigcup_{\mathtt{P} \in \mathrm{PROGS}} [\![\mathtt{P}]\!]_{\mathcal{A}} \supseteq \mathsf{ConsisModelAxioms}\,(\mathcal{A})$, can be derived from the following result: given an abstract execution $\mathcal{X}$ such that $\mathcal{X} \in \mathsf{ConsisModelAxioms}\,(\mathcal{A})$, and for all i,

$$\mathsf{AExectCut}\,(\mathcal{X}, i+1) = \mathsf{UpdateAExec}\left(\mathcal{X}', \mathsf{VIS}_{\mathcal{X}}^{-1}(t), t, \mathcal{T}_{\mathcal{X}}(t)\right)$$

where $\{t\} = \mathsf{AExectCut}\,(\mathcal{X}, i+1) \setminus \mathsf{AExectCut}\,(\mathcal{X}, i)$ and $\mathcal{X}' = \mathsf{AExectCut}\,(\mathcal{X}, i)$. We inductively construct the program $\mathtt{P}$ from the fingerprint $\mathcal{T}_{\mathcal{X}}(t)$ such that

$$(\mathsf{AExectCut}\,(\mathcal{X}, 0), \mathcal{E}_0), \mathtt{P} \xrightarrow{\left(cl, t_1, \mathsf{VIS}^{-1}(t_1), \mathcal{T}_{\mathcal{X}}(t_1)\right)}_{\mathcal{A}} \cdots \xrightarrow{\left(cl, t_i, \mathsf{VIS}^{-1}(t_i), \mathcal{T}_{\mathcal{X}}(t_i)\right)}_{\mathcal{A}}$$
$$(\mathsf{AExectCut}\,(\mathcal{X}, i), \mathcal{E}_i), \mathtt{P}_i \xrightarrow{\left(cl, t_{i+1}, \mathsf{VIS}^{-1}(t_{i+1}), \mathcal{T}_{\mathcal{X}}(t_{i+1})\right)}_{\mathcal{A}} (\mathsf{AExectCut}\,(\mathcal{X}, i+1), \mathcal{E}_{i+1}), \mathtt{P}_{i+1}$$

The full detail is given in appendix A.3 on page 193. ■

5.3 Correspondence to Kv-store Semantics

We explain the connection between abstract executions and kv-stores via dependency graphs. We show the connection between views on the kv-stores and visibility relations on the abstract executions. We then show how to construct an $\mathtt{ET}_{\top}$-trace from an abstract execution, and vice versa. This means that $\mathtt{ET}_{\top}$-traces and abstract executions are equally expressive, which is a key

result for proving the correctness of our definitions of consistency models. In this section, we only consider the traces that do not involve P but only committing fingerprint. In Section 5.4, we will go further and discuss the trace installed with P.

We show how to convert an abstract execution to a dependency graph. The definition of XToD is adapted from [Cerone et al., 2015a; Cerone and Gotsman, 2016]. By the definition of DToK defined in Def. 5.3, we therefore can convert an abstract execution to a kv-store.

Definition 5.15 (Abstract executions to dependency graphs, XToD, and kv-stores, XToK)**.** *Given an abstract execution* $\mathcal{X} \in \mathit{AExecTS}$*, the dependency graph is defined by* $\mathsf{XToD}(\mathcal{X}) \stackrel{def}{=} (T_{\mathcal{X}}, \mathsf{WR}_{\mathcal{X}}, \mathsf{WW}_{\mathcal{X}}, \mathsf{RW}_{\mathcal{X}})$ *where the dependency relations on abstract execution are defined by:*

(1) write-read dependency relation is defined by: $\mathsf{WR}_{\mathcal{X}} \stackrel{def}{=} \bigcup_{k \in \mathit{KEY}} \mathsf{WR}_{\mathcal{X}}(k)$ *where*

$$\mathsf{WR}_{\mathcal{X}}(k) \stackrel{def}{=} \left\{ (t, t') \;\middle|\; \begin{array}{l} t = \mathsf{MaxVisTrans}\left(\mathcal{X}, \mathsf{VIS}_{\mathcal{X}}^{-1}(t'), k\right) \\ \quad \wedge\, \exists v \in \mathit{VALUE}.\, (\mathtt{W}, k, v) \in \mathcal{T}_{\mathcal{X}}(t) \wedge (\mathtt{R}, k, v) \in \mathcal{T}_{\mathcal{X}}(t') \end{array} \right\},$$

(2) write-write dependency relation is defined by: $\mathsf{WW}_{\mathcal{X}} \stackrel{def}{=} \bigcup_{k \in \mathit{KEY}} \mathsf{WW}_{\mathcal{X}}(k)$ *where*

$$\mathsf{WW}_{\mathcal{X}}(k) \stackrel{def}{=} \{(t, t') \mid (t, t') \in \mathsf{AR}_{\mathcal{X}} \wedge \exists v, v' \in \mathit{VALUE}.\, (\mathtt{W}, k, v) \in \mathcal{T}_{\mathcal{X}}(t) \wedge (\mathtt{W}, k, v') \in \mathcal{T}_{\mathcal{X}}(t')\},$$

(3) read-write anti-dependency relation is defined by: $\mathsf{RW}_{\mathcal{X}} \stackrel{def}{=} \bigcup_{k \in \mathit{KEY}} \mathsf{RW}_{\mathcal{X}}(k)$ *where*

$$\mathsf{RW}_{\mathcal{X}}(k) \stackrel{def}{=} \left\{ (t, t') \;\middle|\; \begin{array}{l} \left(\mathsf{MaxVisTrans}\left(\mathcal{X}, \mathsf{VIS}_{\mathcal{X}}^{-1}(t), k\right), t'\right) \in \mathsf{AR}_{\mathcal{X}} \\ \quad \wedge\, \exists v, v' \in \mathit{VALUE}.\, (\mathtt{R}, k, v) \in \mathcal{T}_{\mathcal{X}}(t) \wedge (\mathtt{W}, k, v') \in \mathcal{T}_{\mathcal{X}}(t') \end{array} \right\}.$$

Given DToK *in Def. 5.3, the kv-store induced by the abstract execution* $\mathcal{X}$ *is defined by:*

$$\mathsf{XToK}(\mathcal{X}) = \mathsf{DToK}(\mathsf{XToD}(\mathcal{X})).$$

Each abstract execution $\mathcal{X}$ determines a *unique* and *well-formed* dependency graph, defined by $\mathcal{G} = \mathsf{XToD}(\mathcal{X})$, because: (1) AR is a total order; and (2) $\mathcal{X}$ applies last-write-wins in the sense that a transaction always reads from the latest visible transaction given by MaxVisTrans defined in Def. 5.5. The full proof is given in appendix A.4 on 195. Therefore, $\mathsf{XToK}(\mathcal{X})$ is a uniquely defined and well-formed kv-store, by Prop. A.5. However, abstract executions are *not* bijective to kv-stores in that *several* abstract executions may be encoded to the same kv-store. Because: (1) kv-stores do not have the total arbitration order of transactions, and (2) kv-stores only track the actually WR dependency, in contrast to the *potential write-read dependency* captured by visibility relation.

Apart from the correspondence between individual kv-stores and abstract executions, there is a correspondence between views on kv-stores and visibility relations on abstract executions.

Recall that snapshots in kv-stores are computed from views, while in abstract executions, these snapshots are computed from visibility relations. A kv-store $\mathcal{K}$ is *compatible* with an abstract execution $\mathcal{X}$, written $\mathcal{K} \sim \mathcal{X}$, if and only if any snapshot made on $\mathcal{K}$ can be obtained by an snapshot made on $\mathcal{X}$, and vice versa.

Definition 5.16 (Compatibility between kv-stores and abstract executions)**.** *Given an abstract execution $\mathcal{X}$, the view induced by a set of visible transactions T on $\mathcal{X}$, written* $\mathsf{GetView}\,(\mathcal{X}, T)$*, is defined by:* $\mathsf{GetView}\,(\mathcal{X}, T) \stackrel{def}{=} \lambda k \in \textsc{Key}.\, \{0\} \cup \{i \mid \mathsf{WriterOf}\,(\mathsf{XToK}\,(\mathcal{X})\,(k, i)) \in T\}$*. A kv-store $\mathcal{K}$ and an abstract execution $\mathcal{X}$ are compatible, written $\mathcal{K} \sim \mathcal{X}$, if and only if:*

(1) for any transaction t,

$$t \in \mathcal{K} \Leftrightarrow t \in \mathcal{X} \tag{5.19}$$

(2) for any subset of transactions $T \subseteq \mathcal{X}$ such that $t_0 \in T$,

$$\mathsf{AExecSnapshot}\,(\mathcal{X}, T) = \mathsf{Snapshot}\,(\mathcal{K}, \mathsf{GetView}\,(\mathcal{X}, T)) \tag{5.20}$$

(3) given the definition of VisTrans*, defined in Def. 4.32, for any view $u \in \textsc{ViewOn}\,(\mathcal{K})$ on kv-store $\mathcal{K}$,*

$$\mathsf{Snapshot}\,(\mathcal{K}, u) = \mathsf{AExecSnapshot}\,(\mathcal{X}, \mathsf{VisTrans}\,(\mathcal{K}, u))\,. \tag{5.21}$$

Given a set of transactions T, the GetView function extracts a view that includes all versions written by any transaction in T. A kv-store $\mathcal{K}$ is compatible with an abstract execution $\mathcal{X}$, if and only if: (1) they contain the same set of transactions (Eq. (5.19)); and (2) the snapshot induced by a set of transactions T on $\mathcal{X}$, is the same as the snapshot induced by the view $\mathsf{GetView}\,(\mathcal{X}, T)$ on $\mathcal{K}$ (Eq. (5.20)) and (3) vice versa (Eq. (5.21)).

Theorem 5.17 (Compatibility of $\mathcal{X}$ and $\mathsf{XToK}\,(\mathcal{X})$)**.** *For any abstract execution $\mathcal{X}$, $\mathcal{X} \sim \mathsf{XToK}\,(\mathcal{X})$.*

Proof sketch. It can be derived from the definition of XToK. Given $\mathcal{X}$, let $\mathcal{K} = \mathsf{XToK}\,(\mathcal{X})$. Eq. (5.19) trivial holds. Note that, given an abstract execution $\mathcal{X}$ and a set of transactions T, the view $\mathsf{GetView}\,(\mathcal{X}, T)$ is well-formed on $\mathsf{XToD}\,(\mathcal{X})$ (detail is given in Prop. A.9 on page 196). For Eq. (5.20), it is sufficient to prove

$$\mathsf{AExecSnapshot}\,(\mathcal{X}, T)\,(k) = \mathsf{Snapshot}\,(\mathcal{K}, \mathsf{GetView}\,(\mathcal{X}, T))\,(k).$$

Note that $\mathsf{WW}_{\mathcal{K}} = \mathsf{WW}_{\mathcal{X}} \subseteq \mathsf{AR}_{\mathcal{X}}$. This means if two transactions t, t' both write to key k, and if $t, t' \in T$, then the $\mathsf{AExecSnapshot}\,(\mathcal{X}, T)\,(k)$ may include the value written by t' but not t. Similarly by definition of GetView and Snapshot, the snapshot $\mathsf{Snapshot}\,(\mathcal{K}, \mathsf{GetView}\,(\mathcal{X}, T))\,(k)$

may contain the value of version written by t' but not t. Therefore, we have the prove for Eq. (5.20). Similarly, Eq. (5.21) can be derived by $\mathsf{WW}_{\mathcal{K}} = \mathsf{WW}_{\mathcal{X}} \subseteq \mathsf{AR}_{\mathcal{X}}$. The full proof is given in appendix A.4 on page 195. ■

We now show how to construct an abstract execution from an $\mathtt{ET}_\top$-trace, via $\mathsf{TraceToX}(\tau)$ defined in Def. 5.18. Recall that $\mathtt{ET}_\top$ is defined by $\mathtt{CanCommit}_\top \overset{def}{\Leftrightarrow} \mathtt{true}$ and $\mathtt{ViewShift}_\top \overset{def}{\Leftrightarrow} \mathtt{true}$. The opposite direction is given in Def. 5.20. These two results are the foundation to prove specifies execution test $\mathtt{ET}_M$ for consistency model M is correct with respect to the set of visibility axioms $\mathcal{A}_M$.

Recall that it is enough to only consider normalised traces Theorem 4.30. Given a normalised $\mathtt{ET}_\top$-trace τ,

$$\begin{aligned}\tau = (\mathcal{K}_0,\mathcal{U}_0) \xrightarrow{(cl_1,u_1)}_{\mathtt{ET}_\top} (\mathcal{K}_0,\mathcal{U}'_0) \xrightarrow{(cl_1,\mathcal{F}_1)}_{\mathtt{ET}_\top} (\mathcal{K}_1,\mathcal{U}_1) \xrightarrow{(cl_2,u_2)}_{\mathtt{ET}_\top} (\mathcal{K}_1,\mathcal{U}'_1) \\ \xrightarrow{(cl_2,\mathcal{F}_2)}_{\mathtt{ET}_\top} (\mathcal{K}_2,\mathcal{U}_2) \xrightarrow{(cl_3,u_3)}_{\mathtt{ET}_\top} \cdots \xrightarrow{(cl_n,u_n)}_{\mathtt{ET}_\top} \left(\mathcal{K}_{n-1},\mathcal{U}'_{n-1}\right) \xrightarrow{(cl_n,\mathcal{F}_n)}_{\mathtt{ET}_\top} (\mathcal{K}_n,\mathcal{U}_n)\end{aligned}$$

function $\mathsf{TraceToX}$ inductively constructs a set of possible abstract executions $\mathcal{X}$:

(1) a view-shift transition, (cl_i, u_i) in the trace τ does not affect $\mathcal{X}$; and

(2) a fingerprint transition $(cl_i, \mathcal{F}_i)$ with a given view u_i corresponds to a new transaction node append to $\mathcal{X}$ via $\mathsf{UpdateAExec}$, of which the set of visible transactions, T contains all the writers of the versions include in the view u, that is $\mathsf{VisTrans}(\mathcal{K}',\mathcal{U}(cl))$, and some arbitrary read-only transactions T_{rd}.

It is safe to include any numbers of read-only transactions, because the read-only transactions do not affect the snapshots. By the construction, the arbitrary order in the abstract executions $\mathcal{X} \in \mathsf{TraceToX}(\tau)$ matches the commit order in the trace τ.

Definition 5.18 ($\mathtt{ET}_\top$-traces to abstract executions)**.** *Given an* $\mathtt{ET}_\top$ *trace* τ*, the set of abstract executions, written* $\mathsf{TraceToX}(\tau)$*, is defined by:*

$$\begin{aligned}\mathsf{TraceToX}((\mathcal{K}_0,\mathcal{U}_0)) &\overset{def}{=} (\{t_0 \mapsto \{(k,v_0) \mid k \in \mathit{KEY} \wedge v_0 \in \mathsf{InitialValue}(k)\}\},\emptyset,\emptyset) \\ \mathsf{TraceToX}\left(\tau \xrightarrow{(cl,u)}_\top (\mathcal{K},\mathcal{U})\right) &\overset{def}{=} \mathsf{TraceToX}(\tau) \\ \mathsf{TraceToX}\left(\tau \xrightarrow{(cl,\mathcal{F})}_\top (\mathcal{K},\mathcal{U})\right) &\overset{def}{=} \mathtt{let}\ (\mathcal{K}',\mathcal{U}') = \mathsf{Last}(\tau), t = \mathcal{K} \setminus \mathcal{K}', \\ &\qquad T_{rd} \subseteq \{t' \mid \forall l,k,v.\,(l,k,v) \in \mathcal{T}_{\mathcal{X}}(t') \Rightarrow l = \mathtt{R}\}, \\ &\qquad T = \mathsf{VisTrans}(\mathcal{K}',\mathcal{U}(cl)) \cup T_{rd}\ \mathtt{and}\ \mathcal{X} \in \mathsf{TraceToX}(\tau) \\ &\qquad \mathtt{in}\ \{\mathsf{UpdateAExec}(\mathcal{X},T,\mathcal{F},t)\}.\end{aligned}$$

where $\mathsf{VisTrans}$ *is defined in Def. 4.32.*

Theorem 5.19 (Well-formed abstract executions of XToTrace)**.** *Given an* $\mathtt{ET}_\top$*-trace* τ*, an abstract execution* $\mathcal{X}$ *such that* $\mathcal{X} \in \mathsf{TraceToX}(\tau)$*, is a well-formed abstract execution and* $\mathsf{Last}(\tau)_{|0} = \mathsf{XToK}(\mathcal{X})$*.*

Proof sketch. It is trivial by induction on the length of the trace. For view-shift transition, we directly apply inductive hypothesis. For fingerprint transition, the result can be derived from the following result (Prop. A.10 on page 198): Given: (1) a kv-store $\mathcal{K}$; (2) an abstract execution $\mathcal{X}$ such that $\mathcal{K} = \mathsf{XToK}(\mathcal{X})$; (3) a view $u = \mathsf{GetView}(\mathcal{K}, T)$ for transaction set $T \subseteq \mathcal{X}$ with $t_0 \in T$; and (4) the next abstract execution $\mathcal{X}' = \mathsf{UpdateAExec}(\mathcal{X}, T, \mathcal{F}, t)$ for some fingerprint $\mathcal{F}$ and fresh transaction $t \in \mathsf{NextAExecTxID}(\mathcal{X}, cl)$, then $\mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t) = \mathsf{XToK}(\mathcal{X}')$. The full detail is given in appendix A.4 on 197. ■

Inversely, given an abstract execution $\mathcal{X}$, function $\mathsf{XToTrace}(\mathcal{X})$, defined in Def. 5.20, defines a set of $\mathtt{ET}_\top$-traces in its normalised form such that any such trace τ satisfies $\mathsf{Last}(\tau)_{|0} = \mathsf{XToK}(\mathcal{X})$.

Given an abstract execution $\mathcal{X}$ such that

$$t_0 \xrightarrow{\mathsf{AR}_\mathcal{X}} t_1 \xrightarrow{\mathsf{AR}_\mathcal{X}} \cdots \xrightarrow{\mathsf{AR}_\mathcal{X}} t_n$$

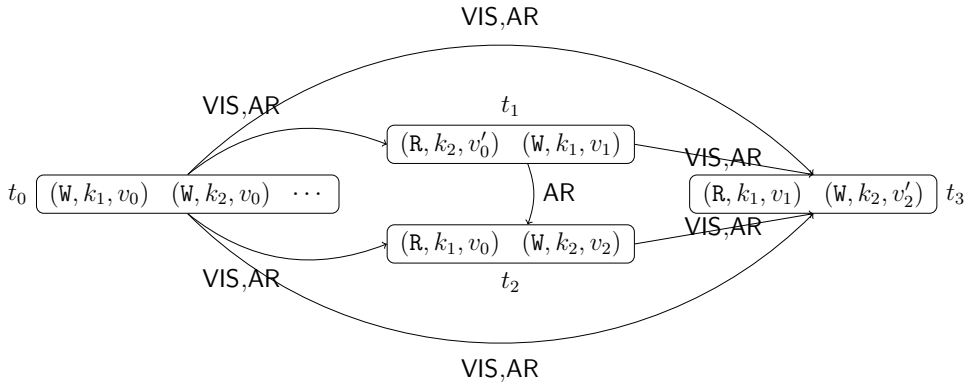
we can construct an $\mathtt{ET}_\top$-trace inductively in the order of AR. In the $(i+1)^{\text{th}}$ step of XToTraceN, that is $\mathsf{XToTraceN}(\mathcal{X}, i+1)$, the next transaction t_i must be the difference between i^{th} and $(i+1)^{\text{th}}$ cut of $\mathcal{X}$. Let cl be the client that commits t_i, and τ be a trace from the result of i^{th} step of XToTraceN. We can construct a new trace by appending a view-shift step and then a fingerprint step:

(1) the view-shift step advances the view of cl to a new view u that extracts from the set of visible transactions of t_i, that is, $u = \mathsf{GetView}\left(\mathcal{X}, \mathsf{VIS}^{-1}(t_i)\right)$;

(2) the fingerprint step simply commits the fingerprint $\mathcal{T}_\mathcal{X}(t_i)$ in the sense that it updates the kv-store to $\mathcal{K}'$ and shift the view u to a new view u' afterwards.

The abstract execution does not contains the precise information about view u', but it can be approximated using ApproxView function, that is: (1) if there is more transactions from the same client cl, the view after commit can be extracted from the intersection of the visible set of the immediate next transaction of cl, that is, $\mathsf{VIS}^{-1}_\mathcal{X}(t^m_{cl})$, and the set of transactions committed before t^n_{cl}, that is, $\left(\mathsf{AR}^{-1}_\mathcal{X}\right)^?(t^n_{cl})$; and (2) otherwise, the view after commit can be any valid view on $\mathcal{K}'$. It is easy to see that ApproxView always returns a set of valid view (the detail is given in Prop. A.11 on page 200).

Definition 5.20 (Abstract executions to $\mathtt{ET}_\top$-traces)**.** *Given an abstract execution* $\mathcal{X} \in$ *AExecTs and an index* $i \in \mathbb{N}$*, the set of* $\mathtt{ET}_\top$*-traces induced by the abstract execution,* $\mathsf{XToTrace}(\mathcal{X})$*, is*



(a) An example $\mathcal{X}$

$$\cdots \xrightarrow{(cl_1, \mathcal{F}_1)} k_1 \mapsto \begin{array}{|c|c|c|c|}\hline v_0 & t_0 & v_1 & t_1 \\ \hline & \emptyset & & \emptyset \\ \hline \end{array} \quad k_2 \mapsto \begin{array}{|c|c|}\hline v_0 & t_0 \\ \hline & \{t_1\} \\ \hline \end{array}$$

(b) An example trace τ containing t_0, t_1

$$\cdots \xrightarrow{(cl_1, \mathcal{F}_1)} k_1 \mapsto \begin{array}{|c|c|c|c|}\hline v_0 & t_0 & v_1 & t_1 \\ \hline & \emptyset & & \emptyset \\ \hline \end{array} \quad k_2 \mapsto \begin{array}{|c|c|}\hline v_0 & t_0 \\ \hline & \{t_1\} \\ \hline \end{array} \xrightarrow{(cl_2, u_2)} k_1 \mapsto \begin{array}{|c|c|c|c|}\hline v_0 & t_0 & v_1 & t_1 \\ \hline & \emptyset & & \emptyset \\ \hline \end{array} \quad k_2 \mapsto \begin{array}{|c|c|}\hline v_0 & t_0 \\ \hline & \{t_1\} \\ \hline \end{array}$$

(c) An example trace τ' containing t_0, t_1 and view-shift of t_2, where $u_2 = \{k_1 \mapsto \{0\}, k_2 \mapsto \{0\}\}$

$$\cdots \xrightarrow{(cl_2, u_2)} k_1 \mapsto \begin{array}{|c|c|c|c|}\hline v_0 & t_0 & v_1 & t_1 \\ \hline & \emptyset & & \emptyset \\ \hline \end{array} \quad k_2 \mapsto \begin{array}{|c|c|}\hline v_0 & t_0 \\ \hline & \{t_1\} \\ \hline \end{array} \xrightarrow{(cl_2, \mathcal{F}_2)} k_1 \mapsto \begin{array}{|c|c|c|c|}\hline v_0 & t_0 & v_1 & t_1 \\ \hline & \{t_2\} & & \emptyset \\ \hline \end{array} \quad k_2 \mapsto \begin{array}{|c|c|c|c|}\hline v_0 & t_0 & v_2 & t_2 \\ \hline & \{t_1\} & & \emptyset \\ \hline \end{array}$$

(d) An example trace τ'' containing t_0, t_1, t_2, where $\mathcal{F}_2 = \{(\mathtt{R}, k_1, v_1), (\mathtt{W}, k_2, v_2')\}$ and $(t_2, t_3) \notin \mathsf{SO}$

$$\cdots \xrightarrow{(cl_2, u_2)} k_1 \mapsto \begin{array}{|c|c|c|c|}\hline v_0 & t_0 & v_1 & t_1 \\ \hline & \emptyset & & \emptyset \\ \hline \end{array} \quad k_2 \mapsto \begin{array}{|c|c|}\hline v_0 & t_0 \\ \hline & \{t_1\} \\ \hline \end{array} \xrightarrow{(cl_2, \mathcal{F}_2)} k_1 \mapsto \begin{array}{|c|c|c|c|}\hline v_0 & t_0 & v_1 & t_1 \\ \hline & \{t_2\} & & \emptyset \\ \hline \end{array} \quad k_2 \mapsto \begin{array}{|c|c|c|c|}\hline v_0 & t_0 & v_2 & t_2 \\ \hline & \{t_1\} & & \emptyset \\ \hline \end{array}$$

(e) An example trace τ'' containing t_0, t_1, t_2, where $\mathcal{F}_2 = \{(\mathtt{R}, k_1, v_1), (\mathtt{W}, k_2, v_2')\}$ and $(t_2, t_3) \in \mathsf{SO}$

Figure 5.6: An example of $\mathsf{XToTrace}(\mathcal{X})$

defined by:

$$\mathsf{XToTrace}(\mathcal{X}) = \mathsf{XToTraceN}(\mathcal{X}, |\mathcal{X}| - 1),$$

where, given definition of AExectCut *in Def. 5.13,* $\mathsf{XToTraceN}(\mathcal{X}, |\mathcal{X}|)$ *is defined by:*

$$\begin{aligned}
\mathsf{XToTraceN}(\mathcal{X}, 0) &\stackrel{def}{=} \{(\mathcal{K}_0, \mathcal{U}_0) \mid \forall cl \in \mathsf{Dom}(\mathcal{U}_0).\, \mathcal{U}_0(cl) = \{0\}\} \\
\mathsf{XToTraceN}(\mathcal{X}, i+1) &\stackrel{def}{=} \mathtt{let}\ \mathcal{X}' = \mathsf{AExectCut}(\mathcal{X}, i), \mathcal{X}'' = \mathsf{AExectCut}(\mathcal{X}, i+1), \\
&\qquad \{t_{cl}^n\} = \mathcal{X}'' \setminus \mathcal{X}', \tau \in \mathsf{XToTraceN}(\mathcal{X}, i), (\mathcal{K}, \mathcal{U}) = \mathsf{Last}(\tau), \\
&\qquad \mathcal{F} = \mathcal{T}_{\mathcal{X}}(t_{cl}^n), \mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t_{cl}^n), \\
&\qquad \mathtt{and}\ u = \mathsf{GetView}\left(\mathcal{X}, \mathsf{VIS}^{-1}(t_{cl}^n)\right)\ \mathtt{in} \\
&\qquad \left\{ \tau \xrightarrow{(cl,u)}_{\mathtt{ET}} (\mathcal{K}, \mathcal{U}[cl \mapsto u]) \xrightarrow{(cl,\mathcal{F})}_{\mathtt{ET}} (\mathcal{K}', \mathcal{U}[cl \mapsto u']) \;\middle|\; u' \in \mathsf{ApproxView}(\mathcal{X}, i+1, t_{cl}^n) \right\}
\end{aligned}$$

$$\mathsf{ApproxView}\,(\mathcal{X}, i, t^n_{cl}) \stackrel{def}{=} \texttt{let}\ \mathcal{X}' = \mathsf{AExectCut}\,(\mathcal{X}, i)\ \texttt{in}$$
$$\begin{cases} \left\{ \mathsf{GetView}\,(\mathcal{X}', T) \;\middle|\; T \subseteq \mathsf{VIS}^{-1}_{\mathcal{X}}\,(t^m_{cl}) \cap (\mathsf{AR}^{-1}_{\mathcal{X}})^{?}(t^n_{cl}) \right\} \\ \qquad\qquad \textit{if}\ t^m_{cl} \in \mathcal{X} \wedge m > n \wedge \forall t. \neg \left(t^n_{cl} \xrightarrow{\mathsf{SO}} t \xrightarrow{\mathsf{SO}} t^m_{cl} \right) \\ \textsc{ViewOn}\,(\mathsf{XToK}\,(\mathcal{X}')) \qquad\qquad \textit{otherwise} \end{cases}$$

Let us consider a concrete example in Fig. 5.6. Given the abstract example depicted in Fig. 5.6a, assume that Fig. 5.6b is a trace τ and its final kv-store after t_0 and t_1. The next transaction is t_2. In the definition of $\mathsf{XToTraceN}$ in Def. 5.20, we append one view-shift and one fingerprint steps to τ for each transaction in abstract execution. The view-shift step (cl_2, u_2) advances the view to a new view $u = \mathsf{GetView}\left(\mathcal{X}, \mathsf{VIS}^{-1}\,(t_2)\right)$, depicted in Fig. 5.6b; this view includes all versions written by the visibility transactions $\mathsf{VIS}^{-1}\,(t_2)$. The fingerprint step $(cl_2, \mathcal{F}_2)$ commits fingerprint $\mathcal{F}_2$ to the kv-store, updating the kv-store to $\mathcal{K}' = \mathsf{UpdateKV}\,(\mathcal{K}, u_2, \mathcal{F}_2, t_2)$; the result kv-store is shown in Figs. 5.6d and 5.6e. Last, the view u to a new view vi'. In the abstract execution, there is no direct information about u', however, it can be approximated via $\mathsf{ApproxView}$:

(1) if there is more transaction t_3 from the same client as t_2, that is, $(t_2, t_3) \in \mathsf{SO}$, then the view u' can be approximated by projecting the visibility transactions of t_3, that is, $u' = \mathsf{GetView}\,(\mathcal{X}', T)$ and $T \subseteq \mathsf{VIS}^{-1}_{\mathcal{X}}\,(t_3) \cap (\mathsf{AR}^{-1}_{\mathcal{X}})^{?}(t_2)$, which guarantees that the u' does not contract with t_3 (Fig. 5.6e);

(2) if there is no more transaction from the same client as t_2 (for example, $(t_2, t_3) \notin \mathsf{SO}$ in Fig. 5.6d) the view u' can be arbitrary (Fig. 5.6d).

Given an abstract execution, any trace τ in $\mathsf{XToTrace}\,(\mathcal{X})$ is a valid $\mathtt{ET}_\top$-trace, and the kv-store $\mathcal{K}$ in the final configuration can be directly extracted from $\mathcal{X}$ in the sense that $\mathcal{K} = \mathsf{XToK}\,(\mathcal{X})$.

Theorem 5.21 (Abstract executions to well-formed $\mathtt{ET}_\top$-traces)**.** *Given an abstract execution $\mathcal{X}$, any trace $\tau \in \mathsf{XToTrace}\,(\mathcal{X})$ is a valid $\mathtt{ET}_\top$-trace and the final kv-store $\mathcal{K}$ such that $(\mathcal{K}, \_) = \mathsf{Last}\,(\tau)$ satisfies that $\mathcal{K} = \mathsf{XToK}\,(\mathcal{X})$.*

Proof sketch. Given the definition of $\mathsf{XToTrace}$, we prove a stronger result that, for any number i, kv-store $\mathcal{K}$, view environment $\mathcal{U}$ and trace τ,

$$\tau = \mathsf{XToTraceN}\,(\mathcal{X}, i) \wedge (\mathcal{K}, \mathcal{U}) = \mathsf{Last}\,(\tau) \wedge \mathcal{K} = \mathsf{XToK}\,(\mathsf{AExectCut}\,(\mathcal{X}, i))\,.$$

This result can be proved by induction on the number i. Note that in each inductive step, there are two more transitions, a view-shift transition and a fingerprint transition. Assume the next transaction is identified by t, from a client cl. For the view-shift transition, we prove that the new view, which is the view induced by the visible transactions of t, contains more versions

than the view of cl. This can be derived from the definition of $\mathsf{ApproxView}$. For the fingerprint transition, we prove the new kv-store $\mathcal{K}$ by committing t is equivalent to the i^{th} cut of $\mathcal{X}$, that is, $\mathcal{K} = \mathsf{XToK}\left(\mathsf{AExectCut}\left(\mathcal{X}, i\right)\right)$. This can be proven by the inductive case of the definition of $\mathsf{XToTrace}$. The full proof is given in appendix A.4 on page 200. ■

5.4 Soundness and Completeness Constructors

We show how all the results illustrated so far can be put together to show that a consistency model using an execution test $\mathtt{ET}$ on kv-stores is *sound and complete* with respect to the declarative definition using visibility axioms $\mathcal{A}$ on abstract. This means that, the set of kv-stores induced by $\mathtt{ET}$ for some program $\mathtt{P}$ is compatible with the set of abstract execution tests induced by $\mathcal{A}$, that is, $\bigcup_{\mathtt{P}\in\mathrm{PROGS}}[\![\mathtt{P}]\!]_{\mathtt{ET}} = \{\mathsf{XToK}\left(\mathcal{X}\right) \mid \mathcal{X} \in \mathsf{ConsisModelAxioms}\left(\mathcal{A}\right)\}$. Note that, our operational semantics talks about the execution of some programs $\mathtt{P}$, while the well-known declarative semantics only talks about fingerprints. We introduce soundness and completeness constructors, which lift the conditions on execution tests $\mathtt{ET}$ and visibility axioms $\mathcal{A}$ with respect to single transitions to the level of traces, and therefore to the level of reachable kv-stores and abstract exactions. Recall that in the abstract execution operational semantics, the client cl loses information of the visible transactions immediately after it commits a transaction t. However, in our kv-store semantics, the view u' after cl commit t must satisfy $\mathtt{ViewShift}$ predicate. In Def. 5.20 when we convert abstract executions to kv-store traces, we can easy approximate u' due to $\mathtt{ViewShift}_{\top} \stackrel{def}{=} \mathtt{true}$. However, for any execution test $\mathtt{ET}$ such that $\mathtt{ViewShift}_{\mathtt{ET}}$ is not always true, We introduce a notation of *abstract execution invariant on the visible transactions* I, which, for each client cl, tracks the minimum set of visible transactions $I\left(\mathcal{X}, cl\right)$ for the next transaction of cl. This invariant I on abstract executions is analogous to the $\mathtt{ViewShift}$ on kv-stores. In Def. 5.23, we use the invariant to define the *soundness constructor* from an execution test $\mathtt{ET}$ to a set of visibility axioms $\mathcal{A}$ in the sense that $\mathtt{ET}$ is sound to $\mathcal{A}$, there must exist an invariant *aexecinv* such that I satisfies some conditions. This soundness is lifted to the level of traces of kv-stores and abstract executions in Theorem 5.24.

Definition 5.22 (Abstract execution invariants for clients)**.** *An* abstract execution invariant on the visible transactions *or just an* invariant *is a function* $I : \mathit{AExecTs} \times \mathit{CID} \to \mathcal{P}\left(\mathit{TxID}\right)$ *such that for any clients* cl*,*

$$I\left(\mathcal{X}, cl\right) \subseteq \mathcal{X}, \tag{5.22}$$

$$\forall T \subseteq \mathcal{X}.\, \forall \mathcal{F} \in \mathit{Fp}.\, \forall i \in \mathbb{N}.\, \forall cl' \in \mathit{CID}.\, \forall t^{i}_{cl'} \in \mathit{TxID}.\, cl \neq cl' \Rightarrow I\left(\mathsf{UpdateAExec}\left(\mathcal{X}, T, \mathcal{F}, t^{i}_{cl'}\right), cl\right) = I\left(\mathcal{X}, cl\right). \tag{5.23}$$

Given a client cl, the invariant $I\left(\mathcal{X}, cl\right)$: (1) must be a subset of transactions in the abstract execution (Eq. (5.22)); and (2) must be stable in the sense that the set does not modify by new transactions committed by different clients cl' (Eq. (5.23)), which is similar to our operational

semantics on kv-stores: cl' cannot modify the view of cl. Give the invariant, we define the *soundness constructor*: that is, the notion of the soundness of an execution test $\texttt{ET}$ with respect to a set visibility axioms $\mathcal{A}$.

Definition 5.23 (Soundness constructor)**.** *An execution test* $\texttt{ET}$ *is* sound *with respect to a set of visibility axioms* $\mathcal{A}$*, if and only if there exists an invariant* I *such that: for any client* cl*, views* u, u'*, transaction identifier* t*, fingerprint* $\mathcal{F}$*, kv-stores* $\mathcal{K}, \mathcal{K}'$*, abstract execution* $\mathcal{X}$ *and set of transactions* T*, assume that:*

(1) $\mathcal{K}$ *describes the same state as* $\mathcal{X}$*, that is,* $\mathcal{K} = \mathsf{XToK}(\mathcal{X})$*;*

(2) cl *is the next scheduled client and* $t \in \mathsf{NextTxID}(\mathcal{K}, cl)$ *is the next transaction;*

(3) $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t)$ *is the new kv-store of committing* t *to* $\mathcal{K}$ *with fingerprint* $\mathcal{F}$ *under view* u*;*

(4) this step satisfies the execution test, $(\mathcal{K}, u) \xrightarrow{\mathcal{F}}_{\texttt{ET}} (\mathcal{K}', u')$*;*

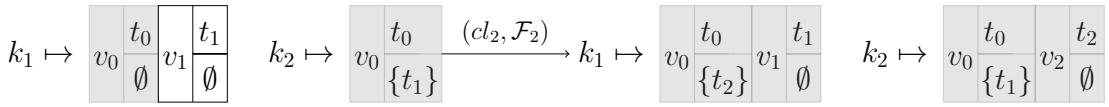
(5) $T = \mathsf{VisTrans}(\mathcal{K}, u)$ *is the set of visible transactions induced by* u *such that* $I(\mathcal{X}, cl) \subseteq T$*;*

then there exists a corresponding new abstract execution $\mathcal{X}'$ *such that*

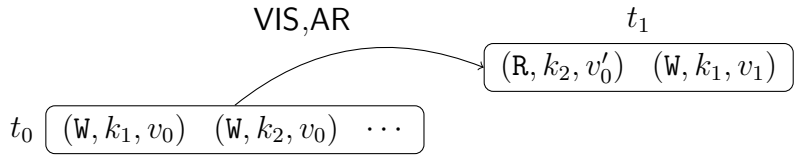
$$\begin{aligned} &\exists T_{rd} \subseteq \mathsf{ReadOnlyTrans}(\mathcal{X}) \,.\, \mathcal{X}' = \mathsf{UpdateAExec}(\mathcal{X}, T \cup T_{rd}, \mathcal{F}, t) \\ &\quad \wedge \forall A \in \mathcal{A}.\, A(\mathcal{X}')^{-1}(t) \subseteq T \cup T_{rd} \qquad (5.24) \\ &\quad \wedge I(\mathcal{X}', cl) \subseteq \mathsf{VisTrans}(\mathcal{K}', u') \qquad (5.25) \end{aligned}$$

where $\mathsf{ReadOnlyTrans}(\mathcal{X}) \stackrel{def}{=} \{t_{rd} \mid \forall l, k, v.\, (l, k, v) \in \mathcal{T}_{\mathcal{X}}(t_{rd}) \Rightarrow l = \texttt{R}\}$ *and* $\mathsf{NextTxID}, \mathsf{UpdateKV}, \mathsf{VisTrans}, \mathsf{UpdateAExec}, \mathsf{XToK}$ *are defined in Defs. 4.19, 4.20, 4.32, 5.10 and 5.15 respectively.*

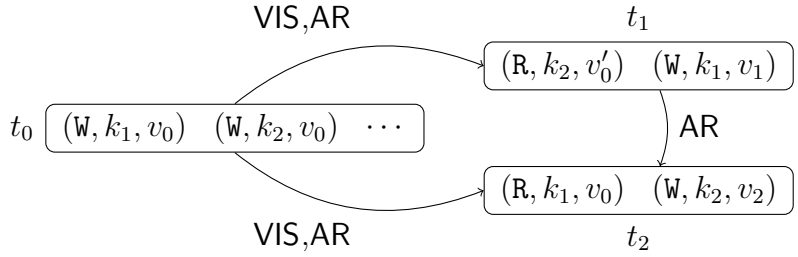
The soundness constructor states that: given an execution test $\texttt{ET}$, there exists an invariant such that for any element $(\mathcal{K}, u, \mathcal{F}, \mathcal{K}', u')$ in $\texttt{ET}$ such that $(\mathcal{K}, u) \xrightarrow{\mathcal{F}}_{\texttt{ET}} (\mathcal{K}', u')$, and any abstract execution $\mathcal{X}$ which can be extracted to $\mathcal{K}$ in the sense that $\mathcal{K} = \mathsf{XToK}(\mathcal{X})$, then there exists a set of read-only transactions T_{rd}, together with the set of visible transactions induced by the view, $T = \mathsf{VisTrans}(\mathcal{K}, u)$, such that the abstract execution $\mathcal{X}$ is updated to $\mathcal{X}'$ by the fingerprint $\mathcal{F}$ and with the visible set of transactions being $T_{\text{rd}} \cup T$, and: (1) this update is allowed by the set of visibility axioms (Eq. (5.24)); and (2) the invariant holds on the new graph $\mathcal{X}'$ (Eq. (5.25)). This constructor means that given an execution test $\texttt{ET}$, if a transaction t is allowed to $\mathcal{K}$ with a given view u, updating the kv-stores and view to $\mathcal{K}'$ and vi', then Let us take Fig. 5.7 as an example. Assume a step on kv-stores under $\texttt{ET}_{\texttt{RYW}}$ depicted in Fig. 5.7a: client cl_2 commit transaction t_2 to the kv-store $\mathcal{K}$ with view u_2, updating them to $\mathcal{K}'$ and u_2' respectively. Then assume an abstract execution $\mathcal{X}$, depicted in Fig. 5.7b, such that $\mathcal{K} = \mathsf{XToK}(\mathcal{X})$. The invariant $I(\mathcal{X}, cl)$ contains all the version written by the client cl, of which in $\mathcal{X}$ we give formal definition



(a) Client cl_2 commit transaction t_2 to the kv-store $\mathcal{K}$ with view u_2, updating them to $\mathcal{K}'$ and u_2' respectively



(b) An example $\mathcal{X}$ such that $\mathcal{K} = \mathsf{XToK}(\mathcal{X})$, where $I(\mathcal{X}, cl_2) = \emptyset$



(c) New abstract execution $\mathcal{X}' = \mathsf{UpdateAExec}(\mathcal{X}, T \cup T_{\mathrm{rd}}, \mathcal{F}_2, t_2)$, where $I(\mathcal{X}, cl_2) \subseteq T = \{t_0, t_1\}$ and $T_{\mathrm{rd}} = \emptyset$

Figure 5.7: An example of soundness constructor

in 94. For example, in Fig. 5.7b, $I(\mathcal{X}, cl_2) = \emptyset$. This means the set of visibility transactions T satisfies $I(\mathcal{X}, cl_2) \subseteq T = \mathsf{VisTrans}(\mathcal{K}, u_2) = \{t_0\}$, as in the hypothesis in Def. 5.23. The Def. 5.23 then states that, if $\mathtt{ET}_{\mathtt{RYW}}$ is sound with respect to the $\mathcal{A}_{\mathtt{RYW}} = \{\lambda\mathcal{X}.\, \mathsf{SO}_{\mathcal{X}}\}$ [Burckhardt et al., 2014], then there exists a set of read-only transactions T_{rd}, picking $T_{\mathrm{rd}} = \emptyset$ here, and a new abstract execution $\mathcal{X}' = \mathsf{UpdateAExec}(\mathcal{X}, T \cup T_{\mathrm{rd}}, \mathcal{F}_2, t_2)$, depicted in Fig. 5.7c, satisfying Eqs. (5.24) and (5.25). Eq. (5.24) states that the new transaction t_2 satisfies the visibility axioms $\mathcal{A}_{\mathtt{RYW}}$, that is, $\mathsf{SO}^{-1}_{\mathcal{X}'}(t_2) = \emptyset \subseteq T$. Eq. (5.25) states that the invariant holds under $\mathcal{X}'$, that is, $I(\mathcal{X}', cl_2) = \{t_2\} \subseteq \mathsf{VisTrans}(\mathcal{K}', u') = \{t_0, t_2\}$. This soundness constructor can be lifted to the level of traces.

Theorem 5.24 (Soundness of execution tests)**.** *Given an execution test* $\mathtt{ET}$ *is sound with respect to a set of visibility axioms* $\mathcal{A}$*, then*

$$\bigcup_{\mathtt{P} \in \mathit{PROGS}} [\![\mathtt{P}]\!]_{\mathtt{ET}} \subseteq \{\mathsf{XToK}(\mathcal{X}) \mid \mathcal{X} \in \mathsf{ConsisModelAxioms}(\mathcal{A})\}.$$

Proof sketch. By Theorem 4.31 stating that $\bigcup_{\mathtt{P} \in \mathrm{PROGS}} = \mathsf{ConsisModel}(\mathtt{ET})$, it suffices to prove the following result

$$\mathsf{ConsisModel}(\mathtt{ET}) \subseteq \{\mathsf{XToK}(\mathcal{X}) \mid \mathcal{X} \in \mathsf{ConsisModelAxioms}(\mathcal{A})\}.$$

Let I be the invariant that gives rise of soundness. We prove a stronger result that, for any

ET-trace τ, there exists an abstract execution $\mathcal{X}$ that satisfies $\mathcal{A}$ and preserves I, that is,

$$\tau = (\mathcal{K}_0, \mathcal{U}_0) \Rightarrow^*_{\texttt{ET}} (\mathcal{K}, \mathcal{U}) \wedge \mathcal{K} = \mathsf{XToK}(\mathcal{X}) \wedge \mathcal{X} \in \mathsf{ConsisModelAxioms}(\mathcal{A}) \\ \wedge \forall cl \in \mathsf{Dom}(\mathcal{U}) \,.\, I(\mathcal{X}, cl) \subseteq \mathsf{VisTrans}(\mathcal{K}, \mathcal{U}(cl))$$

First, the trace τ must be a $\texttt{ET}_\top$-trace, and by Theorem 5.19 for any $\mathcal{X}' \in \mathsf{TraceToX}(\tau)$, it satisfies that $\mathcal{K} = \mathsf{XToK}(\mathcal{X}')$. We now show that we can always find an abstract execution $\mathcal{X} \in \mathsf{TraceToX}(\tau)$ such that $\texttt{AExecSat}(\mathcal{X}, \mathcal{A})$ and $I(\mathcal{X}, cl) \subseteq T_{cl}$ for a client cl, a set of transactions $T_{cl} = \mathsf{VisTrans}(\mathcal{K}, \mathcal{U}(cl)) \cup T_{\mathrm{rd}}$ and a set of read-only transactions T_{rd} in $\mathcal{X}$. We prove it by induction on the length of τ. Note that by Theorem 4.30, it is safe to assume that τ is a normalised trace. The inductive case can be derived by the definition of the soundness constructor. The full proof is given in appendix A.5 on page 202. ■

The completeness constructor is given in Def. 5.25. If an execution test ET is complete with respect to $\mathcal{A}$, if and only if, a transaction step t on abstract execution for a client cl can be converted to an equivalent step on kv-store. We convert the set of visible transactions $\mathsf{VIS}^{-1}_{\mathcal{X}}(t)$ of t into the view u used to update the kv-store, and approximates the view u' after the update. This step on kv-store must satisfy execution test ET.

Definition 5.25 (Complete constructor)**.** *An execution test* ET *is* complete *with respect to a set of visibility axioms $\mathcal{A}$, if and only if, for any abstract executions $\mathcal{X}, \mathcal{X}', \mathcal{X}''$, kv-stores $\mathcal{K}', \mathcal{K}''$, indexes i, n, views u', u'', client cl, transaction identifier t^i_{cl} and set of transactions T such that:*

(1) $\mathcal{X}$ is the final abstract execution;

(2) $\mathcal{X}' = \mathsf{AExectCut}(\mathcal{X}, n)$ and $\mathcal{X}'' = \mathsf{AExectCut}(\mathcal{X}, n+1)$ are the n^{th} and $(n+1)^{th}$ cuts of $\mathcal{X}$ respectively;

(3) $\{t^i_{cl}\} = \mathcal{X}'' \setminus \mathcal{X}'$ is the last transaction in $\mathcal{X}''$, and this transaction is from client cl;

(4) $T = \mathsf{VIS}^{-1}_{\mathcal{X}}(t^i_{cl})$ is the set of visible transactions of t^i_{cl};

(5) $\mathcal{K}' = \mathsf{XToK}(\mathcal{X}')$ and $\mathcal{K}'' = \mathsf{XToK}(\mathcal{X}'')$ describe the same states as $\mathcal{X}'$ and $\mathcal{X}''$, respectively; and

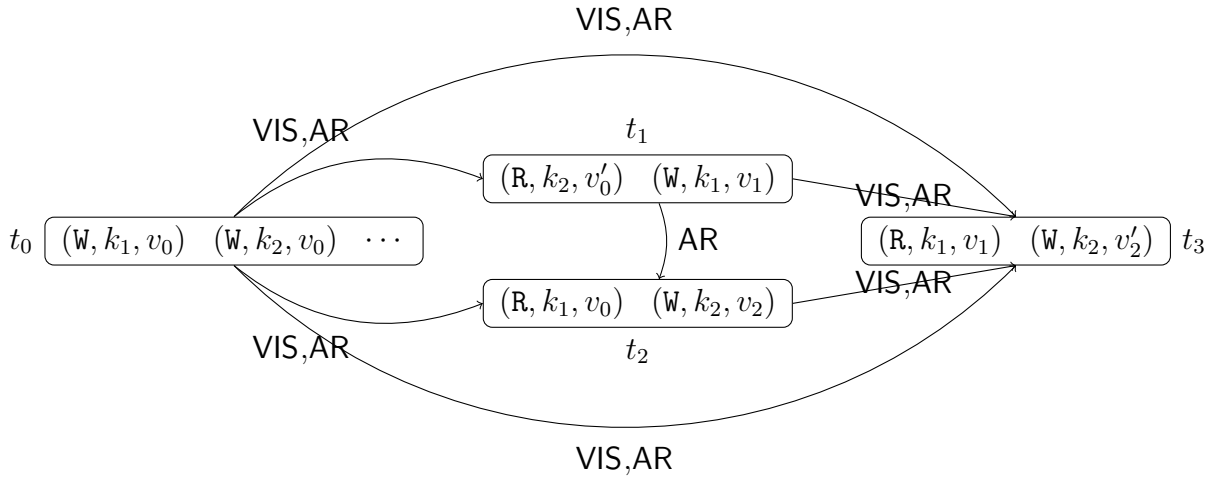
(6) $u = \mathsf{GetView}(\mathcal{K}', T)$ is the view before t^i_{cl};

then there exist a set of transactions T' committed before t^i_{cl}, that is, $(\mathsf{AR}^{-1}_{\mathcal{X}})^?(t^i_{cl})$ such that

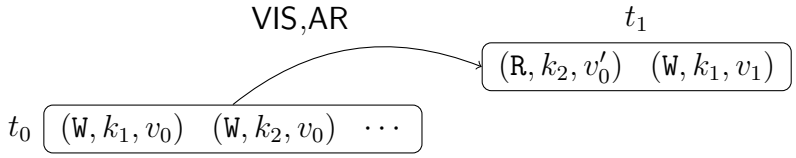
$$\exists t \in \mathcal{X} \,.\, t = \mathsf{Min}_{\mathsf{SO}}\left(\left\{t' \;\middle|\; t' \in \mathcal{X} \wedge (t^i_{cl}, t') \in \mathsf{SO}\right\}\right) \Rightarrow T' \subseteq \mathsf{VIS}^{-1}_{\mathcal{X}}(t) \tag{5.26}$$

$$\wedge\, u' = \mathsf{GetView}(\mathcal{X}, T') \wedge (\mathcal{K}', u) \xrightarrow{\mathcal{T}_{\mathcal{X}}(t^i_{cl})}_{\texttt{ET}} (\mathcal{K}'', u'). \tag{5.27}$$

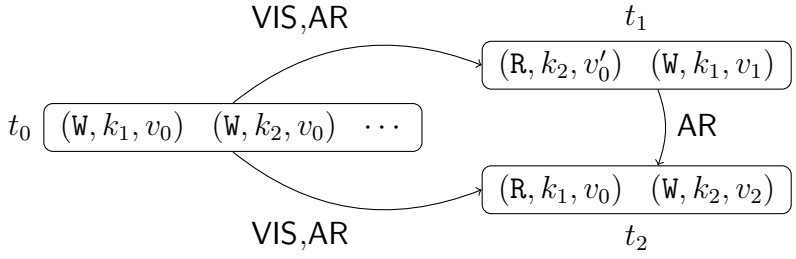
where AExectCut, GetView *are defined in Defs. 5.13 and 5.16.*



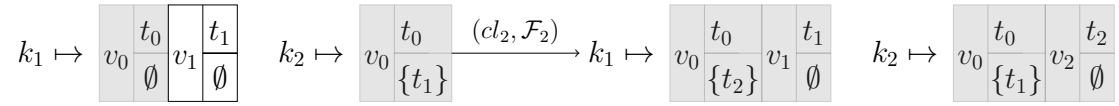
(a) An example $\mathcal{X}$ where $(t_2, t_3) \in SO$



(b) The first cut of $\mathcal{X}' = \mathsf{AExectCut}\,(\mathcal{X}, 1)$



(c) The second cut of $\mathcal{X}'' = \mathsf{AExectCut}\,(\mathcal{X}, 2)$



(d) Transaction step of $txid_2$, updating $\mathcal{K}' = \mathsf{XToK}\,(\mathcal{X}')$ and view $u' = \mathsf{GetView}\left(\mathcal{K}', \mathsf{VIS}^{-1}\,(t_2)\right)$ to new kv-store $\mathcal{K}'' = \mathsf{XToK}\,(\mathcal{X}'')$ and new view $u' = \mathsf{GetView}\,(\mathcal{K}', T')$

Figure 5.8: An example of completeness constructor

For any abstract execution $\mathcal{X}$ and the n^{th} and $(n+1)^{\text{th}}$ cuts of the abstract execution $\mathcal{X}$, that is, $\mathcal{X}'$ and $\mathcal{X}''$ respectively, the complete constructor states that: given the new transaction $\{t^i_{cl}\} = \mathcal{X}'' \setminus \mathcal{X}'$ in the $(n+1)^{\text{th}}$ cut, and the view inducted visible set of the transaction, $u' = \mathsf{GetView}\,(\mathcal{K}', T)$ where $\mathcal{K}' = \mathsf{XToK}\,(\mathcal{X}')$, then there exists an approximation of the view, u'', such that, the fingerprint $\mathcal{T}_{\mathcal{X}}\,(t^i_{cl})$ is committed to the kv-store $\mathcal{K}'$ with the view u', updates the kv-store to $\mathcal{K}'' = \mathsf{XToK}\,(\mathcal{X}'')$ and the view u' to u''. The view u'' is defined by intersection of the set of transactions T' committed before t^i_{cl}, that is, $(\mathsf{AR}^{-1}_{\mathcal{X}})^{?}(t^i_{cl})$, and T' agrees with the next transaction t (if exists) of the same client cl, that is, $T' \subseteq \mathsf{VIS}^{-1}_{\mathcal{X}}\,(t)$ (Eq. (5.26)). This update on kv-store must satisfies $\mathtt{ET}$ (Eq. (5.27)).

Let us take Fig. 5.8 as an example. Assume the final abstract execution $\mathcal{X}$ that satisfies read your write, depicted in Fig. 5.8a. Note that $(t_2, t_3) \in \mathsf{SO}$ in $\mathcal{X}$. We focus on the first and second cuts, $\mathcal{X}'$ and $\mathcal{X}''$, shown in Figs. 5.8b and 5.8c, in which t_2 is the last transaction in $\mathcal{X}''$ with the set of visible transactions being $T = \{t_0\}$. We pick $T' = \{t_0, t_1, t_2\}$ and we now show that it satisfies Eqs. (5.26) and (5.27). Because t_3 is from the same client as t_2 in $\mathcal{X}$, and the set of visible transactions of t_3 is $\mathsf{VIS}^{-1}(t_3) = k\{t_0, t_1, t_2\}$, Eq. (5.27) holds as $T' \subseteq \mathsf{VIS}^{-1}(t_3)$. Given T', Eq. (5.27) holds, that is, $(\mathcal{K}', u) \xrightarrow{\mathcal{T}_{\mathcal{X}}(t_2)}_{\texttt{RYW}} (\mathcal{K}'', u')$ shown in Fig. 5.8d. The completeness constructor can be lifted to the level of traces.

Theorem 5.26 (Completeness of execution tests)**.** *Given an execution test* ET *that is complete with respect to a set of visibility axioms* $\mathcal{A}$*, then* $\{\mathsf{XToK}(\mathcal{X}) \mid \mathcal{X} \in \mathsf{ConsisModelAxioms}(\mathcal{A})\} \subseteq \bigcup_{\mathtt{P} \in PROGS} [\![\mathtt{P}]\!]_{\mathcal{A}}$.

Proof sketch. By Theorem 4.31 stating that $\bigcup_{\mathtt{P} \in \mathrm{PROGS}} [\![\mathtt{P}]\!]_{\mathcal{A}} = \mathsf{ConsisModel}(\mathtt{ET})$, it suffices to prove the following result

$$\{\mathsf{XToK}(\mathcal{X}) \mid \mathcal{X} \in \mathsf{ConsisModelAxioms}(\mathcal{A})\} \subseteq \mathsf{ConsisModel}(\mathtt{ET}).$$

It is sufficient to prove that: for any abstract execution $\mathcal{X} \in \mathsf{ConsisModelAxioms}(\mathcal{A})$, there exists an ET-trace τ such that $\mathsf{Last}(\tau)_{|0} = \mathsf{XToK}(\mathcal{X})$. We prove a stronger result: that is, for any cut of $\mathcal{X}$ such as $\mathcal{X}' = \mathsf{AExectCut}(\mathcal{X}, i)$, there exists an ET-trace τ such that, for any kv-store $\mathcal{K}$, view environment $\mathcal{U}$ and a set of clients CL,

$$CL = \{cl \mid \exists t^n_{cl}.\, t^n_{cl} \in \mathcal{X}\} \wedge (\mathcal{K}, \mathcal{U}) = \mathsf{Last}(\tau) \Rightarrow \mathcal{K} = \mathsf{XToK}(\mathcal{X}') \wedge \left(\begin{array}{l} \forall cl \in CL.\, \forall t \in \mathcal{X}'.\, t = \mathsf{Max}_{\mathsf{AR}_{\mathcal{X}}}(\{t' \mid t \in \mathcal{X}' \wedge t' = t_{\overline{cl}}\}) \Rightarrow \exists T' \subseteq (\mathsf{AR}^{-1}_{\mathcal{X}})^{?}(t). \\ \quad \big(\exists t''.\, t'' = \mathsf{Min}_{\mathsf{SO}}(\{t' \mid t' \in \mathcal{X} \wedge (t, t') \in \mathsf{SO}\}) \Rightarrow T' \subseteq \mathsf{VIS}^{-1}_{\mathcal{X}}(t'')\big) \\ \qquad\qquad \wedge\, \mathcal{U}(cl) = \mathsf{GetView}(\mathcal{X}, T') \end{array} \right).$$

Since $\mathcal{X} \in \mathsf{ConsisModelAxioms}(\mathcal{A})$, then $\mathcal{X} \in \mathsf{ConsisModelAxioms}(\mathcal{A}_{\top})$. By Def. 5.20 and Theorem 5.21, any trace τ in $\mathsf{XToTrace}(\mathcal{X})$ must be a valid $\mathtt{ET}_{\top}$-trace and the kv-store in the final state of τ is compatible with $\mathcal{X}$. We now prove that τ is a ET-trace and preserves the invariant above by induction one the cut i. The inductive case can be derived from the completeness constructor. The full proof is given in appendix A.5 on page 203. ■

5.5 Soundness and Completeness of Execution Tests

We prove all the execution tests in Section 4.3 are sound and complete with the corresponding set of visibility axioms. We first prove Theorem 5.27 stating that the prefix closure property on views on kv-stores matches the closure property of visibility relation on abstract executions. For example, an execution test ET with CanCommit defined by $\mathtt{PreClosed}(\mathcal{K}, u, \mathsf{SO})$ is intuitively sound and complete with respect to the axioms $\mathcal{A} = \{\lambda\mathcal{X}.\, \mathsf{SO}; \mathsf{VIS}_{\mathcal{X}}\}$: both of them requires

that, if a transaction t observes effect of another transaction t' (observation means a view in kv-stores and visible transactions in abstract executions), then t must also observe effect of transactions t'' from the same client as t'. However, ET focuses on versions, that is, transactions that at least has one write operation, while $\mathcal{A}$ covers also read-only transactions. Hence, Eq. (5.28) states that, given a view u such that $\texttt{PreClosed}\,(\mathcal{K}, u, \mathsf{SO})$ and the set of transactions $T = \mathsf{VisTrans}\,(\mathcal{K}, u)$ induced by the view, there exists a set of read-only transactions T_{rd} such that $(T \cup T_{\mathrm{rd}}) \subseteq (\mathsf{SO})^{-1}(T \cup T_{\mathrm{rd}})$. Recall the definition of $\texttt{PreClosed}\,(\mathcal{K}, u, \mathsf{SO})$ in Def. 4.32:

$$\mathsf{VisTrans}\,(\mathcal{K}, u) = \Big((\mathsf{SO}^*)^{-1}(\mathsf{VisTrans}\,(\mathcal{K}, u)) \setminus \mathsf{ReadOnlyTrans}\,(\mathcal{K})\Big),$$

which means that $T \cup T_{\mathrm{rd}} = (\mathsf{SO}^*)^{-1}(T) \subseteq (\mathsf{SO}^*)^{-1}(T \cup T_{\mathrm{rd}})$, and therefore $(T \cup T_{\mathrm{rd}}) \subseteq (\mathsf{SO})^{-1}(T \cup T_{\mathrm{rd}})$. This means that, if u is used to commit a transaction t to kv-stores, the equivalent steps in abstract executions must satisfy $\mathcal{A}$. One possible set of visible transactions of t is $T \cup T_{\mathrm{rd}}$: $(\mathsf{SO}; \mathsf{VIS}_{\mathcal{X}})^{-1}\,(t) = \mathsf{SO}^{-1}\,(T \cup T_{\mathrm{rd}}) \subseteq T \cup T_{\mathrm{rd}}$. The inverse from $\mathcal{A}$ to ET is also true, described in Eq. (5.29).

Theorem 5.27 (View closure equal to visibility closure)**.** *Assume $\mathcal{K}$ and $\mathcal{X}$ such that $\mathcal{K} = \mathsf{XToK}\,(\mathcal{X})$. Assume relations $R_{\mathcal{K}}$ and $R_{\mathcal{X}}$ such that $R_{\mathcal{K}} = R_{\mathcal{X}}$. For any transaction t and fingerprint $\mathcal{F}$, view $u \in \textsc{ViewOn}\,(\mathcal{K})$ and set of transaction $T \subseteq \mathcal{X}$,*

$$\begin{aligned}
&\texttt{PreClosed}\,(\mathcal{K}, u, R_{\mathcal{K}}) \wedge T = \mathsf{VisTrans}\,(\mathcal{K}, u) \\
&\quad \Rightarrow \exists T_{rd} \subseteq \mathsf{ReadOnlyTrans}\,(\mathcal{X})\,.\,(T \cup T_{rd}) \subseteq R_{\mathcal{X}}^{-1}(T \cup T_{rd}), && (5.28)\\
&T \subseteq R_{\mathcal{X}}^{-1}(T) \wedge u = \mathsf{GetView}\,(\mathcal{X}, T) \Rightarrow \texttt{PreClosed}\,(\mathcal{K}, u, R_{\mathcal{K}})\,. && (5.29)
\end{aligned}$$

Proof. We prove Eqs. (5.28) and (5.29) respectively.

(1) [**Case**: **Eq. (5.28)**] Let $T = \mathsf{VisTrans}\,(\mathcal{K}, u)$. Let $T_{\mathrm{rd}} = (R_{\mathcal{K}}^*)^{-1}(\mathsf{VisTrans}\,(\mathcal{K}, u)) \setminus T$ be the set of read-only transactions. Recall the definition of PreClosed in Def. 4.32:

$$\begin{aligned}
&T = \mathsf{VisTrans}\,(\mathcal{K}, u) = \\
&\quad \big((R_{\mathcal{K}}^*)^{-1}(\mathsf{VisTrans}\,(\mathcal{K}, u)) \setminus \{t \mid t \in \mathcal{K} \wedge \forall k.\, \forall i.\, t \neq \mathsf{WriterOf}\,(\mathcal{K}\,(k, i))\}\big)\,.
\end{aligned}$$

It follows that $T = \big((R_{\mathcal{K}}^*)^{-1}(T) \setminus T_{\mathrm{rd}}\big)$. By $R_{\mathcal{K}} = R_{\mathcal{X}}$, we have $T \cup T_{\mathrm{rd}} = (R_{\mathcal{X}}^*)^{-1}(T)$. For any transaction $t_{\mathrm{rd}} \in T_{\mathrm{rd}}$, it must be the case that $(t_{\mathrm{rd}}, t) \in (R_{\mathcal{K}}^*)$ for some transaction $t \in T$; therefore $T \cup T_{\mathrm{rd}} = (R_{\mathcal{X}}^*)^{-1}(T \cup T_{\mathrm{rd}})$.

(2) [**Case**: **Eq. (5.29)**] Let $T' = \mathsf{VisTrans}\,(\mathcal{K}, u)$ be the set of visible transactions and $T_{\mathrm{rd}} = \{t \mid t \in \mathcal{K} \wedge \forall k.\, \forall i.\, t \neq \mathsf{WriterOf}\,(\mathcal{K}\,(k, i))\}$ be the set of all read-only transactions. Since $u = \mathsf{GetView}\,(\mathcal{X}, T)$ then $T' \subseteq T \setminus T_{\mathrm{rd}}$. By the definition PreClosed, it suffices to prove the following result,

$$T' = (R_{\mathcal{K}}^*)^{-1}(T') \setminus T_{\mathrm{rd}}$$

It is trivially $T' \subseteq \left((R^*_{\mathcal{K}})^{-1}(T') \setminus T_{\text{rd}}\right)$ then it remains to prove that $\left((R^*_{\mathcal{K}})^{-1}(T') \setminus T_{\text{rd}}\right) \subseteq T'$. Since $R^*_{\mathcal{K}} = \bigcup_{i\in\mathbb{N}} R^i_{\mathcal{K}}$, We now prove the following result:

$$\forall i \in \mathbb{N}. \left((R^i_{\mathcal{K}})^{-1}(T') \setminus T_{\text{rd}}\right) \subseteq T'$$

by induction on i.

(i) [**Base Case**: $i = 0$] Because $T' = T \setminus T_{\text{rd}}$, it is trivial that $\left((R^0_{\mathcal{K}})^{-1}(T') \setminus T_{\text{rd}}\right) \subseteq T'$ and $(R^0_{\mathcal{K}})^{-1}(T') \subseteq T$.

(ii) [**Inductive Case**: $i > 0$] By the inductive hypothesis, we have $\left((R^{i-1}_{\mathcal{K}})^{-1}(T') \setminus T_{\text{rd}}\right) \subseteq T'$. We now consider $\left((R^i_{\mathcal{K}})^{-1}(T') \setminus T_{\text{rd}}\right) \subseteq T'$. Assume two transactions t, t' such that $t \in T'$ and $(t', t) \in R^i_{\mathcal{K}}$. Note that t must have write since $t \in T'$. There must exist t'' such that $t' \xrightarrow{R_{\mathcal{K}}} t'' \xrightarrow{R^{i-1}_{\mathcal{K}}} t$. By inductive hypothesis, $t'' \in T$. Since $R_{\mathcal{K}} = R_{\mathcal{X}}$ and $T \subseteq R^{-1}_{\mathcal{X}}(T)$, then $t' \in T$. If t' wrote to a key, then $t' \in T'$. Otherwise t' is a read only transaction and $t' \in T_{\text{rd}}$. Thus $\left((R^i_{\mathcal{K}})^{-1}(T') \setminus T_{\text{rd}}\right) \subseteq T'$. ■

We prove all the execution tests in Section 4.3 are sound and complete with the corresponding set of visibility axioms:

`MR`	page 92	`RYW`	page 94	`MW`	page 95	`WFR`	page 95	`CC`	page 96
`UA`	page 99	`CP`	page 99	`PSI`	page 102	`SI`	page 104	`SER`	page 105

We prove that the four sessions guarantees, `MR`, `RYW`, `MW` and `WFR` are sound and complete with respect to the axiomatic definitions presented in [Burckhardt et al., 2014] (the definitions in Fig. 22 in the appendix on page 37). For the well-known consistency models, including `CC`, `PSI`, `SI` and `SER`: (1) we first construct the definitions containing the *minimum* visibility relation from the definitions presented in [Cerone et al., 2015a] using the method presented in [Cerone and Gotsman, 2016]; and we prove that these models are sound and complete with respect to these definitions with minimum visibility.

Monotonic Read The execution test $\texttt{ET}_{\texttt{MR}}$ is sound with respect to the axiomatic definition $\mathcal{A}_{\texttt{MR}} \stackrel{def}{=} \{\lambda\mathcal{X}.\mathsf{VIS}_{\mathcal{X}}; \mathsf{SO}\})$ [Burckhardt et al., 2014]. By Defs. 5.22 and 5.23, we choose the invariant as the following,

$$I_{\texttt{MR}}(\mathcal{X}, cl) \stackrel{def}{=} \bigcup_{t^i_{cl} \in \mathcal{X}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^i_{cl}) \setminus \mathsf{ReadOnlyTrans}(\mathcal{X}).$$

We prove that $\texttt{ET}_{\texttt{MR}}$ is **sound** respective to $\mathcal{A}_{\texttt{MR}}$. Assume two kv-stores $\mathcal{K}, \mathcal{K}'$, two views u, u' and a transaction t for a client cl, a fingerprint $\mathcal{F}$ such that: (1) $(\mathcal{K}, u) \xrightarrow{\mathcal{F}}_{\texttt{MR}} (\mathcal{K}', u')$; (2) $t \in \mathsf{NextTxID}(\mathcal{K}, cl)$; and (3) the new kv-sotre $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t)$. Assume an

abstract execution $\mathcal{X}$ such that $\mathcal{K} = \mathsf{XToK}(\mathcal{X})$. Let $T = \mathsf{VisTrans}(\mathcal{K}, u)$ be the set of visible transactions and T_{rd} be a set of read-only transactions defined by:

$$T_{\mathrm{rd}} = \bigcup_{t^i_{cl} \in \mathcal{X}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^i_{cl}) \cap \mathsf{ReadOnlyTrans}(\mathcal{X}).$$

Let $\mathcal{X}' = \mathsf{UpdateAExec}(\mathcal{X}, T \cup T_{\mathrm{rd}}, \mathcal{F}, t)$ be the new abstract execution. We now check if $\mathcal{X}'$ satisfies Eqs. (5.24) and (5.25) in Def. 5.23.

(1) [**Case**: $\forall A \in \mathcal{A}_{\mathtt{MR}}.\, A(\mathcal{X}')^{-1}(t) \subseteq T \cup T_{\mathbf{rd}}$] By the definition of $\mathcal{A}_{\mathtt{MR}}$, suppose transactions t', t'' such that $t' \xrightarrow{\mathsf{VIS}_{\mathcal{X}'}} t'' \xrightarrow{\mathsf{SO}} t$. Note that $t', t'' \in \mathcal{X}$. Consider t'. If t' is read-only transaction, then $t' \in T_{\mathrm{rd}}$. If t' has write, then $t' \in I_{\mathtt{MR}}(\mathcal{X}, cl)$. Since $I_{\mathtt{MR}}(\mathcal{X}, cl) \subseteq T$, therefore $t' \in T$.

(2) [**Case**: $I_{\mathtt{MR}}(\mathcal{X}', cl) \subseteq \mathsf{VisTrans}(\mathsf{XToK}(\mathcal{X}'), u')$] Because $(\mathcal{K}, u) \xrightarrow{\mathcal{F}}_{\mathtt{MR}} (\mathcal{K}', u')$, it must be the case that $u \sqsubseteq u'$ and thus $T = \mathsf{VisTrans}(\mathcal{K}, u) = \mathsf{VisTrans}(\mathcal{K}', u) \subseteq \mathsf{VisTrans}(\mathcal{K}', u')$. Also, because $\mathcal{K} = \mathsf{XToK}(\mathcal{X})$ and $\mathcal{K}' = \mathsf{XToK}(\mathcal{X}')$, thus $T \subseteq \mathsf{VisTrans}(\mathsf{XToK}(\mathcal{X}'), u')$. Note that $\{t^i_{cl} \mid t^i_{cl} \in \mathcal{X}'\} = \{t^i_{cl} \mid t^i_{cl} \in \mathcal{X}\} \cup t$. Last, we obtain

$$\begin{aligned} I_{\mathtt{MR}}(\mathcal{X}', cl) &= \bigcup_{t^i_{cl} \in \mathcal{X}'} \mathsf{VIS}^{-1}_{\mathcal{X}'}(t^i_{cl}) \setminus \mathsf{ReadOnlyTrans}(\mathcal{X}') \\ &= \bigcup_{t^i_{cl} \in \mathcal{X}} \mathsf{VIS}^{-1}_{\mathcal{X}}(t^i_{cl}) \cup T \setminus \mathsf{ReadOnlyTrans}(\mathcal{X}) \\ &= I_{\mathtt{MR}}(\mathsf{XToK}(\mathcal{X}), u) \cup T \\ &= T \subseteq \mathsf{VisTrans}(\mathsf{XToK}(\mathcal{X}'), u') \end{aligned}$$

Given the two cases above, we proof the soundness of $\mathtt{ET}_{\mathtt{MR}}$. ■

The execution test $\mathtt{ET}_{\mathtt{MR}}$ is complete with respect to the axiomatic definition $\mathcal{A}_{\mathtt{MR}} \stackrel{def}{=} \{\lambda\mathcal{X}.\mathsf{VIS}_{\mathcal{X}}; \mathsf{SO}\})$ [Burckhardt et al., 2014]. We prove that $\mathtt{ET}_{\mathtt{MR}}$ is **complete** respective to $\mathcal{A}_{\mathtt{MR}}$. Assume an abstract execution $\mathcal{X}$ that satisfies $\mathcal{A}_{\mathtt{MR}}$ and a transaction $t \in \mathcal{X}$. Assume i^{th} and $(i+1)^{\mathrm{th}}$ cuts, $\mathcal{X}' = (\mathcal{X}, i)$ and $\mathcal{X}'' = (\mathcal{X}, i+1)$. Let $t^n_{cl} = \mathcal{X}'' \setminus \mathcal{X}'$ be the next transaction, $T = \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl})$ be the set of visible transactions, and $u = \mathsf{GetView}(\mathcal{X}, T)$ be the view with respect to T. Consider if there are more transactions from the client cl in $\mathcal{X}$ after t^n_{cl}.

(1) [**Case**: $(t^n_{cl}, t') \in \mathsf{SO}$ **for** $t' \in \mathcal{X}$] Let the transaction $t = \mathsf{Min}_{\mathsf{SO}}(\{t' \mid (t^n_{cl}, t') \in \mathsf{SO} \wedge t' \in \mathcal{X}'\})$. For this case, let view $u' = \mathsf{GetView}\left(\mathcal{X}, (\mathsf{AR}^{-1}_{\mathcal{X}})^?(t^n_{cl}) \cap \mathsf{VIS}^{-1}_{\mathcal{X}}(t)\right)$. By $\mathcal{A}_{\mathtt{MR}}$, it follows that $T = \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl}) \subseteq \mathsf{VIS}^{-1}_{\mathcal{X}}(t)$, and therefore $u \sqsubseteq u'$ which implies $\mathtt{ET}_{\mathtt{MR}}$.

(2) [**Case**: $\neg((t^n_{cl}, t') \in \mathsf{SO})$] For this case, let view $u' = \mathsf{GetView}\left(\mathcal{X}, (\mathsf{AR}^{-1}_{\mathcal{X}})^?(t^n_{cl})\right)$; this trivially implies $u \sqsubseteq u'$ and thus $\mathtt{ET}_{\mathtt{MR}}$. ■

Read Your Write The execution test $\mathtt{ET}_{\mathtt{RYW}}$ is sound with respect to the axiomatic definition $\mathcal{A}_{\mathtt{RYW}} = \{\lambda\mathcal{X}.\,\mathsf{SO}_{\mathcal{X}}\}$ [Burckhardt et al., 2014]. We pick the following invariant:

$$I_{\mathtt{RYW}}\left(\mathcal{X}, cl\right) \stackrel{def}{=} \bigcup_{t^i_{cl} \in \mathcal{X}} \left(\mathsf{SO}^{-1}\right)^{?}\left(t^i_{cl}\right) \setminus \mathsf{ReadOnlyTrans}\left(\mathcal{X}\right).$$

We prove that $\mathtt{ET}_{\mathtt{RYW}}$ is **sound** respective to $\mathcal{A}_{\mathtt{RYW}}$. Assume two kv-stores $\mathcal{K}, \mathcal{K}'$, two views u, u' and a transaction t for a client cl, a fingerprint $\mathcal{F}$ such that: (1) $(\mathcal{K}, u) \xrightarrow{\mathcal{F}}_{\mathtt{RYW}} (\mathcal{K}', u')$; (2) $t \in \mathsf{NextTxID}\left(\mathcal{K}, cl\right)$; and (3) the new kv-sotre $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t)$. Assume an abstract execution $\mathcal{X}$ such that $\mathcal{K} = \mathsf{XToK}\left(\mathcal{X}\right)$. Let $T = \mathsf{VisTrans}\left(\mathcal{K}, u\right)$ be the set of visible transactions and T_{rd} be a set of read-only transactions defined by:

$$T_{\mathrm{rd}} = \bigcup_{t^i_{cl} \in \mathcal{X}} \left(\mathsf{SO}^{-1}\right)^{?}\left(t^i_{cl}\right) \cap \mathsf{ReadOnlyTrans}\left(\mathcal{X}\right).$$

Let $\mathcal{X}' = \mathsf{UpdateAExec}\left(\mathcal{X}, T \cup T_{\mathrm{rd}}, \mathcal{F}, t\right)$ be the new abstract execution. We now check if $\mathcal{X}'$ satisfies Eqs. (5.24) and (5.25) in Def. 5.23.

(1) [**Case**: $\forall A \in \mathcal{A}.\, A(\mathcal{X}')^{-1}(t) \subseteq T \cup T_{\mathbf{rd}}$] Suppose transactions t, t' such that $t, t' \in \mathcal{X}$ and $(t', t) \in \mathsf{SO}$. If t' is a read-only transaction, $t' \in T_{\mathrm{rd}}$. Otherwise, t' has write, by the definition of $I_{\mathtt{RYW}}$, it follows that $t' \in I_{\mathtt{RYW}}\left(\mathcal{X}, cl\right)$ and therefore $t' \in T$.

(2) [**Case**: $I\left(\mathcal{X}', cl\right) \subseteq \mathsf{VisTrans}\left(\mathsf{XToK}\left(\mathcal{X}'\right), u'\right)$] Because $(\mathcal{K}, u) \xrightarrow{\mathcal{F}}_{\mathtt{RYW}} (\mathcal{K}', u')$, it must be that:

$$\forall k, i.\, \left(\mathsf{WriterOf}\left(\mathcal{K}'(k, i)\right), t\right) \in \mathsf{SO}^{?} \Rightarrow i \in u'(k)$$

and therefore

$$\forall t'.\, \exists k, v.\, (\mathtt{W}, k, v) \in \mathcal{X}'(t') \wedge (t', t) \in \mathsf{SO}^{?} \Rightarrow t' \in \mathsf{VisTrans}\left(\mathsf{XToK}\left(\mathcal{X}'\right), u'\right).$$

Note that $\bigcup_{t^i_{cl} \in \mathcal{X}'} \left(\mathsf{SO}^{-1}\right)^{?}\left(t^i_{cl}\right) = \left(\mathsf{SO}^{-1}\right)^{?}(t)$. Therefore, we have

$$\begin{aligned} I\left(\mathcal{X}', cl\right)_{\mathtt{RYW}} &= \bigcup_{t^i_{cl} \in \mathcal{X}'} \mathsf{SO}^{-1}\left(t^i_{cl}\right) \setminus \mathsf{ReadOnlyTrans}\left(\mathcal{X}\right) \\ &= \left(\left(\mathsf{SO}^{-1}\right)^{?}(t)\right) \setminus \mathsf{ReadOnlyTrans}\left(\mathcal{X}\right) \\ &\subseteq \mathsf{VisTrans}\left(\mathsf{XToK}\left(\mathcal{X}'\right), u'\right) \end{aligned}$$

Given the two cases above, we proof the soundness of $\mathtt{ET}_{\mathtt{RYW}}$. ■

The execution test $\mathtt{ET}_{\mathtt{RYW}}$ is complete with respect to the axiomatic definition $\mathcal{A}_{\mathtt{RYW}} = \{\lambda\mathcal{X}.\,\mathsf{SO}_{\mathcal{X}}\}$ [Burckhardt et al., 2014]. We prove that $\mathtt{ET}_{\mathtt{RYW}}$ is **complete** respective to $\mathcal{A}_{\mathtt{RYW}}$. Assume an abstract execution $\mathcal{X}$ that satisfies $\mathcal{A}_{\mathtt{RYW}}$ and a transaction $t \in \mathcal{X}$. Assume i^{th} and $(i+1)^{\mathrm{th}}$ cuts, $\mathcal{X}' = (\mathcal{X}, i)$ and $\mathcal{X}'' = (\mathcal{X}, i+1)$. Let $t^n_{cl} = \mathcal{X}'' \setminus \mathcal{X}'$ be the next transaction, $T = \mathsf{VIS}^{-1}_{\mathcal{X}}\left(t^n_{cl}\right)$ be the set of visible transactions, and $u = \mathsf{GetView}\left(\mathcal{X}, T\right)$ be the view with respect to T. We construct the final view u' depending on whether t^n_{cl} is the last transaction for the client cl.

(1) [**Case**: $(t^n_{cl}, t') \in \mathsf{SO}$ **for** $t' \in \mathcal{X}$] Let the transaction $t = \mathsf{Min}_{\mathsf{SO}}(\{t' \mid (t^n_{cl}, t') \in \mathsf{SO} \wedge t' \in \mathcal{X}'\})$. For this case, let view $u' = \mathsf{GetView}\left(\mathcal{X}, (\mathsf{AR}^{-1}_{\mathcal{X}})^{?}(t^i_{cl}) \cap \mathsf{VIS}^{-1}_{\mathcal{X}}(t)\right)$. By $\mathcal{A}_{\mathtt{RYW}}$, it follows that, for any transaction t', if $(t', t^i_{cl}) \in \mathsf{SO}^{?}$, then $t' \in \mathsf{VIS}^{-1}_{\mathcal{X}}(t)$. Since $\mathsf{SO} \subseteq \mathsf{AR}$, we know that $t' \in (\mathsf{AR}^{-1}_{\mathcal{X}})^{?}(t^i_{cl}) \cap \mathsf{VIS}^{-1}_{\mathcal{X}}(t)$. Therefore, for any version $\mathcal{K}'(k, j)$ such that $(\mathsf{WriterOf}(\mathcal{K}'(k, j)), t) \in \mathsf{SO}^{?}$, then $j \in u'(k)$.

(2) [**Case**: $\neg((t^n_{cl}, t') \in \mathsf{SO})$] For this case, let $u' = \mathsf{GetView}\left(\mathcal{X}, (\mathsf{AR}^{-1}_{\mathcal{X}})^{?}(t^i_{cl})\right)$ be the final view. It is easy to see that u' satisfies `RYW`. ■

Monotonic Write and Write Follows Read The execution test $\mathtt{ET}_{\mathtt{MW}}$ is sound with respect to the axiomatic definition [Burckhardt et al., 2014],

$$\mathcal{A}_{\mathtt{MW}} = \{\lambda\mathcal{X}.(\mathsf{SO} \cap \mathsf{WW}_{\mathcal{X}}); \mathsf{VIS}_{\mathcal{X}}\})$$

We pick the invariant as empty set, $I(\mathcal{X}, cl) = \emptyset$, since there is no constraint on the view after update.

We prove that $\mathtt{ET}_{\mathtt{MW}}$ is **sound** respective to $\mathcal{A}_{\mathtt{MW}}$. Assume two kv-stores $\mathcal{K}, \mathcal{K}'$, two views u, u' and a transaction t for a client cl, a fingerprint $\mathcal{F}$ such that: (1) $(\mathcal{K}, u) \xrightarrow{\mathcal{F}}_{\mathtt{MW}} (\mathcal{K}', u')$; (2) $t \in \mathsf{NextTxID}(\mathcal{K}, cl)$; and (3) the new kv-sotre $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t)$. Assume an abstract execution $\mathcal{X}$ such that $\mathcal{K} = \mathsf{XToK}(\mathcal{X})$. Let $T = \mathsf{VisTrans}(\mathcal{K}, u)$ be the set of visible transactions and T_{rd} be a set of read-only transactions defined by:

$$T_{\mathrm{rd}} \subseteq \mathsf{ReadOnlyTrans}(\mathcal{X}).$$

Let $\mathcal{X}' = \mathsf{UpdateAExec}(\mathcal{X}, T \cup T_{\mathrm{rd}}, \mathcal{F}, t)$ be the new abstract execution. We now check if $\mathcal{X}'$ satisfies Eqs. (5.24) and (5.25) in Def. 5.23. For Eq. (5.24) and by the definition of $\mathsf{UpdateAExec}$, it suffices to prove that there exists a set of read-only transaction T_{rd} such that

$$(T \cup T_{\mathrm{rd}}) \subseteq R^{-1}_{\mathtt{MW}}(T \cup T_{\mathrm{rd}}). \tag{5.30}$$

Recall that $\mathtt{CanCommit}_{\mathtt{MW}}(\mathcal{K}, u, \mathcal{F}) = \mathtt{PreClosed}(\mathcal{K}, u, R_{\mathtt{MW}})$ such that $R_{\mathtt{MW}} = \mathsf{SO} \cap \mathsf{WW}_{\mathcal{K}}$. Note that $\mathsf{WW}_{\mathcal{K}} = \mathsf{WW}_{\mathcal{X}}$. By Eq. (5.28) and Theorem 5.27, then Eq. (5.30) holds. Eq. (5.25) is trivially true since $I(\mathcal{X}, cl) = \emptyset$. ■

The execution test $\mathtt{ET}_{\mathtt{MW}}$ is complete with respect to the axiomatic definition [Burckhardt et al., 2014],

$$\mathcal{A}_{\mathtt{MW}} = \{\lambda\mathcal{X}.(\mathsf{SO} \cap \mathsf{WW}_{\mathcal{X}}); \mathsf{VIS}_{\mathcal{X}}\})$$

We prove that $\mathtt{ET}_{\mathtt{MW}}$ is **complete** respective to $\mathcal{A}_{\mathtt{MW}}$. Assume an abstract execution $\mathcal{X}$ that satisfies $\mathcal{A}_{\mathtt{MW}}$ and a transaction $t \in \mathcal{X}$. Assume i^{th} and $(i+1)^{\mathrm{th}}$ cuts, $\mathcal{X}' = (\mathcal{X}, i)$ and $\mathcal{X}'' = (\mathcal{X}, i+1)$. Let $t^n_{cl} = \mathcal{X}'' \setminus \mathcal{X}'$ be the next transaction, $T = \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl})$ be the set of visible transactions, and $u = \mathsf{GetView}(\mathcal{X}, T)$ be the view with respect to T. It sufficient to prove

that $\texttt{PreClosed}\,(\mathcal{K}, u, R_{\texttt{MW}})$. Note that $u = \mathsf{GetView}\,(\mathcal{X}, T) = \mathsf{GetView}\,(\mathcal{X}', T)$. Therefore by Eq. (5.29) and Theorem 5.27, then $\texttt{PreClosed}\,(\mathcal{K}, u, R_{\texttt{MW}})$ holds. ■

The soundness and completeness of our definition of WFR with respect to the axiomatic definition, $\mathcal{A}_{\texttt{MW}} = \{\lambda\mathcal{X}.\mathsf{WR}_{\mathcal{X}}; (\mathsf{SO} \cap \mathsf{RW}_{\mathcal{X}}); \mathsf{VIS}_{\mathcal{X}}\})$, can be prove directly using Theorem 5.27. ■

Causal Consistency The widely accepted definition for causal consistency is that VIS is transitive and $\mathsf{SO} \subseteq \mathsf{VIS}$ on abstract executions [Cerone et al., 2015a]. Yet it is for the sack of elegant definition, while there is a equivalent minimum visibility relation (Theorem 5.28) defined by: $\mathcal{A}_{\texttt{CC}} \stackrel{def}{=} \{\lambda\mathcal{X}.\,(\mathsf{WR}_{\mathcal{X}} \cup \mathsf{SO}); \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}, \lambda\mathcal{X}.\,\mathsf{SO} \subseteq \mathsf{VIS}_{\mathcal{X}}\}$, where $\mathsf{WR}_{\mathcal{X}}$ is defined in Def. 5.15.

Theorem 5.28 (Minimum visibility relation for CC)**.** *For two abstract executions $\mathcal{X}, \mathcal{X}'$, the following constrain on visibility,*

$$(\mathsf{WR}_{\mathcal{X}} \cup \mathsf{SO}); \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}} \wedge \mathsf{SO} \subseteq \mathsf{VIS}_{\mathcal{X}} \tag{5.31}$$

is equivalent to

$$\mathsf{VIS}_{\mathcal{X}'}; \mathsf{VIS}_{\mathcal{X}'} \subseteq \mathsf{VIS}_{\mathcal{X}'} \wedge \mathsf{SO} \subseteq \mathsf{VIS}_{\mathcal{X}'} \tag{5.32}$$

in that $\forall t \in \mathit{TxID}.\, \forall\mathcal{F}.\, (\mathcal{F} = \mathcal{T}_{\mathcal{X}}\,(t) \Leftrightarrow \mathcal{F} = \mathcal{X}'(t)) \wedge \mathsf{AR}_{\mathcal{X}} = \mathsf{AR}_{\mathcal{X}'}$.

Proof. For an abstract execution $\mathcal{X}$ that satisfies Eq. (5.31), by Lemma 5.29, there exists $\mathcal{X}'$ that satisfies Eq. (5.32). Assume an abstract execution $\mathcal{X}'$ that satisfies Eq. (5.32). Since $\mathsf{WR}_{\mathcal{X}'} \subseteq \mathsf{VIS}_{\mathcal{X}'}$ by the definition of $\mathsf{WR}_{\mathcal{X}'}$, thus $\mathcal{X}'$ satisfies Eq. (5.31). ■

Lemma 5.29 (Minimum visibility relation for CC)**.** *For any abstract execution $\mathcal{X}$, if it satisfies $\mathcal{A}_{\texttt{CC}}$, there exists a new abstract execution $\mathcal{X}'$ such that* $\mathsf{SO} \in \mathsf{VIS}_{\mathcal{X}}$ *and*

$$\forall t \in \mathit{TxID}.\, \forall\mathcal{F}.\, (\mathcal{F} = \mathcal{T}_{\mathcal{X}}\,(t) \Leftrightarrow \mathcal{F} = \mathcal{X}'(t)) \wedge \mathsf{AR}_{\mathcal{X}} = \mathsf{AR}_{\mathcal{X}'} \wedge \mathsf{VIS}_{\mathcal{X}'}; \mathsf{VIS}_{\mathcal{X}'} \subseteq \mathsf{VIS}_{\mathcal{X}'}. \tag{5.33}$$

Proof. We erase some visibility relation for each transaction following the arbitration order AR until the visibility is transitive. Intuitively, the final visibility relation is exactly $(\mathsf{WR}_{\mathcal{X}} \cup \mathsf{SO})^{+}$. Assume the i^{th} transaction t_i with respect to the arbitration order. Let R_i be a new visibility for the transaction t_i such that $R_{i|2} = \{t_i\}$ for all indexes i and the union of visibility relations $\bigcup_{0 \le j \le i} R_i$ is transitive. We preserve that, for each index i, cut of abstract execution $\mathcal{X}' = \mathsf{AExectCut}\,(\mathcal{X}, i)$ and visibility relation $\mathsf{VIS}' = \bigcup_{0 \le j \le i} R_j$, the following invariant holds:

$$\mathsf{VIS}'; \mathsf{VIS}' \subseteq \mathsf{VIS}', \tag{5.34}$$

$$\forall t \in \mathcal{X}.\, (t, t_i) \in R_i \Rightarrow (t, t_i) \in (\mathsf{WR}_{\mathcal{X}'} \cup \mathsf{SO}). \tag{5.35}$$

We prove the above by induction on the number i.

(1) [**Base Case**: $i = 0$] By the definition of $\mathsf{AExectCut}$, we know that $\mathcal{X}_0 = \mathsf{AExectCut}\,(\mathcal{X}, 0)$ and Eqs. (5.34) and (5.35) trivially hold.

(2) [**Inductive Case**: $i > 0$] Suppose that, for the $(i-1)^{\text{th}}$ step, the abstract execution $\mathcal{X}'' = \mathsf{AExectCut}\,(\mathcal{X}, i-1)$ and the visibility relation $\mathsf{VIS}'' = \bigcup_{0 \leq j \leq i-1} R_j$ satisfy Eqs. (5.34) and (5.35). Let consider i^{th} step, the transaction t_i, the cut $\mathcal{X}' = \mathsf{AExectCut}\,(\mathcal{X}, i)$ and the visibility relation $\mathsf{VIS}' = \bigcup_{0 \leq j \leq i} R_j$. Initially we take R as an empty set. First, we include $\{(t, t_i) \mid (t, t_i) \in \mathsf{WR}_{\mathcal{X}}\}$ to R and, by the definition of $\mathsf{WR}_{\mathcal{X}}$, it trivially does not affect any read operation for the transaction t_i. Then we do the same for SO as that we include $\{(t, t_i) \mid (t, t_i) \in \mathsf{SO}\}$ to R. Note that SO cannot affect any read operation for the transaction t_i neither, otherwise it contradicts to that $\mathsf{SO} \subseteq \mathsf{VIS}_{\mathcal{X}}$ and the definition of $\mathsf{WR}_{\mathcal{X}}$.

For relations $R' = R; \bigcup_{0 \leq j \leq i-1} R_j$ and then $R_i = R \cup R'$, it easy to see that $R \in \mathsf{VIS}_{\mathcal{X}}$ and, then by inductive hypothesis, $R' \in \mathsf{VIS}_{\mathcal{X}}$. We prove that the R_i does not affect any read operation for the transaction t_i by contradiction. Assume distinct transactions t, t' such that $t'' \xrightarrow{R \cup R'} t' \xrightarrow{R \cup R'} t_i$, and immediately by the definition of R and R', then $t'' \xrightarrow{R'} t' \xrightarrow{R} t_i$. Assume that t'' change the read operation for a key k in t_i. This means that there exists a transaction t^* such that $(t^*, t_i) \in \mathsf{WR}_{\mathcal{X}}(k)$ and $(t^*, t'') \in \mathsf{AR}_{\mathcal{X}}$, where the latter implies that $(t'', t_i) \in \mathsf{WR}_{\mathcal{X}}(k)$; there is a contradiction and thus R_i does not affect any read operation for the transaction t_i.

We now prove that Eqs. (5.34) and (5.35) still hold.

(i) [**Case**: **Eq. (5.34)**] Assume a relation $R^* = \bigcup_{0 \leq j \leq i-1} R_j$ and transactions t, t', t'' such that

$$t \xrightarrow{R^* \cup R_i} t' \xrightarrow{R^* \cup R_i} t''.$$

If $t \xrightarrow{R^*} t' \xrightarrow{R^*} t''$, then by inductive hypothesis, $t \xrightarrow{R^*} t''$. Note that $t \xrightarrow{R_i} t' \xrightarrow{R^*} t''$ cannot happen, because it contradict to that $t' = t_i$ and $(t'', t_i) \in \mathsf{AR}_{\mathcal{X}}$. Thus consider $t \xrightarrow{R^*} t' \xrightarrow{R_i} t''$. It must be the case that $t'' = t_i$ and by the definition of R_i, we know that $t \xrightarrow{R_i} t''$.

(ii) [**Case**: **Eq. (5.35)**] By the construction, Eq. (5.35) hold. ■

We pick the invariant as $I_{\mathtt{CC}} = I_{\mathtt{MR}} \cup I_{\mathtt{RYW}}$. We prove that $\mathtt{ET}_{\mathtt{CC}}$ is **sound** respective to $\mathcal{A}_{\mathtt{CC}}$. Assume two kv-stores $\mathcal{K}, \mathcal{K}'$, two views u, u' and a transaction t for a client cl, a fingerprint $\mathcal{F}$ such that: (1) $(\mathcal{K}, u) \xrightarrow{\mathcal{F}}_{\mathtt{CC}} (\mathcal{K}', u')$; (2) $t \in \mathsf{NextTxID}\,(\mathcal{K}, cl)$; and (3) the new kv-sotre $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t)$. Assume an abstract execution $\mathcal{X}$ such that $\mathcal{K} = \mathsf{XToK}\,(\mathcal{X})$. Let $T = \mathsf{VisTrans}\,(\mathcal{K}, u)$ be the set of visible transactions and T_{rd} be a set of read-only transactions defined by:

$$T_{\text{rd}} \supseteq \bigcup_{\{t^i_{cl} \mid t^i_{cl} \in \mathcal{X}\}} \left(\mathsf{VIS}^{-1}_{\mathcal{X}}\,(t^i_{cl}) \cup (\mathsf{SO}^{-1})^?\,(t^i_{cl})\right) \cap \mathsf{ReadOnlyTrans}\,(\mathcal{X})\,.$$

Let $\mathcal{X}' = \mathsf{UpdateAExec}(\mathcal{X}, T \cup T_{\mathrm{rd}}, \mathcal{F}, t)$ be the new abstract execution. We now check if $\mathcal{X}'$ satisfies Eqs. (5.24) and (5.25) in Def. 5.23. Assume

$$T'_{\mathrm{rd}} = \bigcup_{\left\{t^i_{cl} \mid t^i_{cl} \in \mathcal{X}\right\}} \left(\mathsf{VIS}^{-1}_{\mathcal{X}}\left(t^i_{cl}\right) \cup \left(\mathsf{SO}^{-1}\right)^{?}\left(t^i_{cl}\right)\right) \cap \mathsf{ReadOnlyTrans}\left(\mathcal{X}\right).$$

and $T''_{\mathrm{rd}} = T_{\mathrm{rd}} \setminus T'_{\mathrm{rd}}$. We construct T''_{rd} in the following proof. By the definition of soundness, we prove the following results:

$$\mathsf{SO}^{-1}(t) \subseteq T \cup T'_{\mathrm{rd}} \tag{5.36}$$

$$((\mathsf{WR}_{\mathcal{X}'} \cup \mathsf{SO}); \mathsf{VIS}_{\mathcal{X}'})^{-1}(t) \subseteq T \cup T'_{\mathrm{rd}} \cup T''_{\mathrm{rd}} \tag{5.37}$$

$$I_{\mathsf{cc}}\left(\mathcal{X}', cl\right) \subseteq \mathsf{VisTrans}\left(\mathsf{XToK}\left(\mathcal{X}'\right), u'\right) \tag{5.38}$$

Eq. (5.36) can be proven in the same way as in monotonic read on page 92 We now prove Eq. (5.37). Initially we take T''_{rd} to be an empty set. Recall $\mathcal{X}' = \mathsf{UpdateAExec}(\mathcal{X}, T \cup T_{\mathrm{rd}}, \mathcal{F}, t)$. Note that $\mathsf{VIS}^{-1}_{\mathcal{X}'}(t) = T \cup T'_{\mathrm{rd}} \cup T''_{\mathrm{rd}}$. By Eq. (5.28) in Theorem 5.27, there exists T''_{rd} such that $T \cup T''_{\mathrm{rd}}$ is closed in the relation $\mathsf{WR}_{\mathcal{X}'} \cup \mathsf{SO}$. Now consider a transaction $t_{\mathrm{rd}} \in T'_{\mathrm{rd}}$ and assume a transaction t' such that $t' \xrightarrow{\mathsf{WR}_{\mathcal{X}'} \cup \mathsf{SO}} t_{\mathrm{rd}}$. There are two cases depending on t_{rd}.

(1) [**Case**: $t_{\mathbf{rd}} \xrightarrow{\mathsf{VIS}_{\mathcal{X}'}} t'' \xrightarrow{\mathsf{SO}} t$ **for some** t''] For this case, we have

$$\begin{aligned} t' \xrightarrow{\mathsf{WR}_{\mathcal{X}'} \cup \mathsf{SO}} t_{\mathrm{rd}} \xrightarrow{\mathsf{VIS}_{\mathcal{X}'}} t'' \xrightarrow{\mathsf{SO}} t &\Rightarrow t' \xrightarrow{\mathsf{WR}_{\mathcal{X}} \cup \mathsf{SO}} t_{\mathrm{rd}} \xrightarrow{\mathsf{VIS}_{\mathcal{X}}} t'' \xrightarrow{\mathsf{SO}} t \\ &\Rightarrow t' \xrightarrow{\mathsf{VIS}_{\mathcal{X}}} t'' \xrightarrow{\mathsf{SO}} t. \end{aligned}$$

By $I_{\mathtt{MR}}$, we know that $t' \in I_{\mathtt{MR}} \cup T'_{\mathrm{rd}}$.

(2) [**Case**: $t_{\mathbf{rd}} \xrightarrow{\mathsf{SO}} t$] For this case, we have

$$\begin{aligned} t' \xrightarrow{\mathsf{WR}_{\mathcal{X}'} \cup \mathsf{SO}} t_{\mathrm{rd}} \xrightarrow{\mathsf{SO}} t &\Rightarrow t' \xrightarrow{\mathsf{WR}_{\mathcal{X}} \cup \mathsf{SO}} t_{\mathrm{rd}} \xrightarrow{\mathsf{SO}} t \\ &\Rightarrow t' \xrightarrow{\mathsf{VIS}_{\mathcal{X}}} t'' \xrightarrow{\mathsf{SO}} t. \end{aligned}$$

By $I_{\mathtt{MR}}$, we know that $t' \in I_{\mathtt{MR}} \cup T'_{\mathrm{rd}}$.

This means that $T \cup T'_{\mathrm{rd}} \cup T''_{\mathrm{rd}} = T \cup T_{\mathrm{rd}}$ is closed in the relation $(\mathsf{WR}_{\mathcal{X}'} \cup \mathsf{SO})$. Last, Eq. (5.38) can be proven in the same way as in monotonic read on page 92 and read your write one page 94. ■

We prove that $\mathtt{ET}_{\mathsf{cc}}$ is **complete** respective to $\mathcal{A}_{\mathsf{cc}}$. Assume an abstract execution $\mathcal{X}$ that satisfies $\mathcal{A}_{\mathsf{cc}}$ and a transaction $t \in \mathcal{X}$. Assume i^{th} and $(i+1)^{\mathrm{th}}$ cuts, $\mathcal{X}' = (\mathcal{X}, i)$ and $\mathcal{X}'' = (\mathcal{X}, i+1)$. Let $t^n_{cl} = \mathcal{X}'' \setminus \mathcal{X}'$ be the next transaction, $T = \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl})$ be the set of visible transactions, and $u = \mathsf{GetView}(\mathcal{X}, T)$ be the view with respect to T. By Theorem 5.28, it is sufficient to prove with respect to the following visibility axioms, $\mathcal{A}'_{\mathsf{cc}} \stackrel{def}{=} \{\lambda\mathcal{X}.\, \mathsf{VIS}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}, \lambda\mathcal{X}.\, \mathsf{SO} \subseteq \mathsf{VIS}_{\mathcal{X}}\}$.

By the definition of $\texttt{ET}_{\texttt{CC}}$, we prove $\texttt{CanCommit}_{\texttt{CC}}$ and $\texttt{ViewShift}_{\texttt{MR}\cup\texttt{RYW}}$ respectively. Since $(\mathsf{WR}_{\mathcal{X}} \cup \mathsf{SO}); \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$, then $\texttt{CanCommit}_{\texttt{CC}}$ can be derived from Theorem 5.27 and Eq. (5.29) and $\texttt{ViewShift}_{\texttt{RYW}}$ can be proven in the same way as in read your write on page 94. By $\mathsf{VIS}_{\mathcal{X}}; \mathsf{SO} \subseteq \mathsf{VIS}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$, $\texttt{ViewShift}_{\texttt{MR}}$ can be proven in the same way as in monotonic read on page 92.

Update Atomic The execution test $\texttt{ET}_{\texttt{UA}}$ is sound with respect to the axiomatic definition $\mathcal{A}_{\texttt{UA}} = \{\lambda\mathcal{X}.\mathsf{WW}_{\mathcal{X}}\}$. We pick the invariant as $I_{\texttt{UA}}(\mathcal{X}, cl) = \emptyset$, given the fact of no constraint on the final view. We prove that $\texttt{ET}_{\texttt{UA}}$ is **sound** respective to $\mathcal{A}_{\texttt{UA}}$. Assume two kv-stores $\mathcal{K}, \mathcal{K}'$, two views u, u' and a transaction t for a client cl, a fingerprint $\mathcal{F}$ such that: (1) $(\mathcal{K}, u) \xrightarrow{\mathcal{F}}_{\texttt{UA}} (\mathcal{K}', u')$; (2) $t \in \mathsf{NextTxID}(\mathcal{K}, cl)$; and (3) the new kv-sotre $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t)$. Assume an abstract execution $\mathcal{X}$ such that $\mathcal{K} = \mathsf{XTok}(\mathcal{X})$. Let $T = \mathsf{VisTrans}(\mathcal{K}, u)$ be the set of visible transactions and T_{rd} be a set of read-only transactions defined by:

$$T_{\text{rd}} \subseteq \mathsf{ReadOnlyTrans}(\mathcal{X}).$$

Let $\mathcal{X}' = \mathsf{UpdateAExec}(\mathcal{X}, T \cup T_{\text{rd}}, \mathcal{F}, t)$ be the new abstract execution. We now check if $\mathcal{X}'$ satisfies Eqs. (5.24) and (5.25) in Def. 5.23. Because the invariant is empty set, it remains to prove the following result:

$$\mathsf{WW}^{-1}_{\mathcal{X}'}(t) \subseteq T \cup T_{\text{rd}}.$$

Assume a transaction t' such that $(t', t) \in \mathsf{WW}_{\mathcal{X}'}(k)$ for some key k. By $\mathsf{WW}_{\mathcal{X}'} = \mathsf{WW}_{\mathcal{K}'}$, we have $(t', t) \in \mathsf{WW}_{\mathcal{K}'}(k)$. Because t' is a transaction already existing in $\mathcal{K}$, then we have that $\mathsf{WriterOf}(\mathcal{K}(k, i)) = t'$ for some index i. Since t also wrote to the key k, that is, $(\texttt{W}, k, v) \in \mathcal{F}$ for some value v. By the definition of $\texttt{ET}_{\texttt{UA}}$, we know that $i \in u(k)$. Recall that $T = \mathsf{VisTrans}(\mathcal{K}, u)$. This means $t' \in T$. ■

The execution test $\texttt{ET}_{\texttt{UA}}$ is complete with respect to the axiomatic definition $\mathcal{A}_{\texttt{UA}} = \{\lambda\mathcal{X}.\mathsf{WW}_{\mathcal{X}}\}$. We prove that $\texttt{ET}_{\texttt{UA}}$ is **complete** respective to $\mathcal{A}_{\texttt{UA}}$. Assume an abstract execution $\mathcal{X}$ that satisfies $\mathcal{A}_{\texttt{UA}}$ and a transaction $t \in \mathcal{X}$. Assume i^{th} and $(i+1)^{\text{th}}$ cuts, $\mathcal{X}' = (\mathcal{X}, i)$ and $\mathcal{X}'' = (\mathcal{X}, i+1)$. Let $t^n_{cl} = \mathcal{X}'' \setminus \mathcal{X}'$ be the next transaction, $T = \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl})$ be the set of visible transactions, and $u = \mathsf{GetView}(\mathcal{X}, T)$ be the view with respect to T. Let consider a key k that have been overwritten by the transaction t^n_{cl}. By the visibility axiom $\mathsf{WW}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$, for any transaction t that writes to the same key k and was committed before t^n_{cl}, they must be included in the visible set, that is, $t \in \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl})$. Note that $t \xrightarrow{\mathsf{WW}_{\mathcal{X}}} t^n_{cl} \Rightarrow t \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t^n_{cl} \Rightarrow t \in \mathsf{XTok}(\mathcal{X}')$. Since the transaction t wrote to the key k, it means that $\mathsf{WriterOf}(\mathsf{XTok}(\mathcal{X}')(k, j)) = t$ for some index j and then by the definition of $u = \mathsf{GetView}(\mathcal{X}, T)$, we have $j \in u(k)$, which implies $\texttt{CanCommit}_{\texttt{UA}}(\mathsf{XTok}(\mathcal{X}'), u, \mathcal{F})$. ■

Consistency Prefix An abstract execution $\mathcal{X}$ satisfies consistency prefix ($\texttt{CP}$), if it satisfies $\mathsf{AR}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$ and $\mathsf{SO} \subseteq \mathsf{VIS}_{\mathcal{X}}$ [Bernardi and Gotsman, 2016]. Cerone and Gotsman [2016]

proposed a method that can compute and express AR and VIS using $\mathsf{WR}, \mathsf{WW}, \mathsf{RW}$ and SO.

Theorem 5.30 (Minimum visibility relation for ($\mathtt{CP}$))**.** *For any abstract execution $\mathcal{X}$, if it satisfies*

$$\left((\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}}); \mathsf{RW}_{\mathcal{X}}{}^{?} \cup \mathsf{WW}_{\mathcal{X}}\right); \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}} \qquad \mathsf{SO} \subseteq \mathsf{VIS}_{\mathcal{X}} \tag{5.39}$$

then there exists a new $\mathcal{X}'$ such that $\mathcal{T}_{\mathcal{X}}(t) = \mathcal{X}'(t)$ for all $t \in \mathsf{Dom}(\mathcal{X})$, and

$$\mathsf{AR}_{\mathcal{X}'}; \mathsf{VIS}_{\mathcal{X}'} \subseteq \mathsf{VIS}_{\mathcal{X}'} \qquad \mathsf{SO} \subseteq \mathsf{VIS}_{\mathcal{X}'} \tag{5.40}$$

and vice versa.

Proof. We construct $\mathcal{X}$ from $\mathcal{X}'$ and vice versa.

(1) [**Case**: $\mathcal{X}$ **to** $\mathcal{X}'$] Assuming an abstract execution $\mathcal{X}$ that satisfies Eq. (5.39). We use the same method presented in [Cerone and Gotsman, 2016]. We solve the following inequalities (the first five are universally true for all abstract execution and the last two are for $\mathtt{CP}$)

$$\mathsf{WR} \subseteq \mathsf{VIS} \qquad \mathsf{WW} \subseteq \mathsf{AR} \qquad \mathsf{VIS} \subseteq \mathsf{AR} \qquad \mathsf{VIS}; \mathsf{RW} \subseteq \mathsf{AR} \qquad \mathsf{AR}; \mathsf{AR} \subseteq \mathsf{AR}$$
$$\mathsf{SO} \subseteq \mathsf{VIS} \qquad \mathsf{AR}; \mathsf{VIS} \subseteq \mathsf{VIS}$$

The visibility and arbitration relations can be defined by

$$\mathsf{AR} \stackrel{def}{=} \left((\mathsf{SO} \cup \mathsf{WR}); \mathsf{RW}^{?} \cup \mathsf{WW} \cup R\right)^{+} \tag{5.41}$$
$$\mathsf{VIS} \stackrel{def}{=} \left((\mathsf{SO} \cup \mathsf{WR}); \mathsf{RW}^{?} \cup \mathsf{WW} \cup R\right)^{*}; (\mathsf{SO} \cup \mathsf{WR}) \tag{5.42}$$

for some relation $R \subseteq \mathsf{AR}$. We take R such that it extends the relation Eq. (5.41) to a total order and thus we have a new $\mathcal{X}'$ that satisfies Eq. (5.40).

(2) [**Case**: $\mathcal{X}'$ **to** $\mathcal{X}$] Assume an abstract execution $\mathcal{X}'$ that satisfies Eq. (5.40). We have

$$\begin{aligned}\left((\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}}); \mathsf{RW}_{\mathcal{X}}{}^{?} \cup \mathsf{WW}_{\mathcal{X}}\right); \mathsf{VIS}_{\mathcal{X}} \subseteq \left((\mathsf{VIS}_{\mathcal{X}'} \cup \mathsf{VIS}_{\mathcal{X}'}); \mathsf{AR}_{\mathcal{X}'}{}^{?} \cup \mathsf{VIS}_{\mathcal{X}'}\right); \mathsf{VIS}_{\mathcal{X}} \\ \mathsf{VIS}_{\mathcal{X}'}; \mathsf{VIS}_{\mathcal{X}'} \\ \mathsf{AR}_{\mathcal{X}'}; \mathsf{VIS}_{\mathcal{X}'} \\ \mathsf{VIS}_{\mathcal{X}'}\end{aligned}$$

Therefore, the abstract execution $\mathcal{X}'$ satisfies Eq. (5.39). ■

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

$$\mathcal{A}_{\mathtt{CP}} \stackrel{def}{=} \left\{\lambda\mathcal{X}.\, \left((\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}}); \mathsf{RW}_{\mathcal{X}}{}^{?} \cup \mathsf{WW}_{\mathcal{X}}\right); \mathsf{VIS}_{\mathcal{X}}, \lambda\mathcal{X}.\, \mathsf{SO}\right\}.$$

We pick the invariant $I_{\texttt{CP}} = I_{\texttt{MR}} \cup I_{\texttt{RYW}}$. We prove that $\texttt{ET}_{\texttt{CP}}$ is **sound** respective to $\mathcal{A}_{\texttt{CP}}$. Assume two kv-stores $\mathcal{K}, \mathcal{K}'$, two views u, u' and a transaction t for a client cl, a fingerprint $\mathcal{F}$ such that: (1) $(\mathcal{K}, u) \xrightarrow{\mathcal{F}}_{\texttt{CP}} (\mathcal{K}', u')$; (2) $t \in \mathsf{NextTxID}(\mathcal{K}, cl)$; and (3) the new kv-sotre $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t)$. Assume an abstract execution $\mathcal{X}$ such that $\mathcal{K} = \mathsf{XToK}(\mathcal{X})$. Let $T = \mathsf{VisTrans}(\mathcal{K}, u)$ be the set of visible transactions and T_{rd} be a set of read-only transactions defined by:

$$T_{\text{rd}} \supseteq \bigcup_{\left\{t^i_{cl} \mid t^i_{cl} \in \mathcal{X}\right\}} \left(\mathsf{VIS}^{-1}_{\mathcal{X}}(t^i_{cl}) \cup \left(\mathsf{SO}^{-1}\right)^{?}(t^i_{cl})\right) \setminus \mathsf{ReadOnlyTrans}(\mathcal{X}).$$

Let $\mathcal{X}' = \mathsf{UpdateAExec}(\mathcal{X}, T \cup T_{\text{rd}}, \mathcal{F}, t)$ be the new abstract execution. We now check if $\mathcal{X}'$ satisfies Eqs. (5.24) and (5.25) in Def. 5.23. Assume

$$T'_{\text{rd}} = \bigcup_{\left\{t^i_{cl} \mid t^i_{cl} \in \mathcal{X}\right\}} \left(\mathsf{VIS}^{-1}_{\mathcal{X}}(t^i_{cl}) \cup \left(\mathsf{SO}^{-1}\right)^{?}(t^i_{cl})\right) \cap \mathsf{ReadOnlyTrans}(\mathcal{X}).$$

and $T''_{\text{rd}} = T_{\text{rd}} \setminus T'_{\text{rd}}$. We construct T''_{rd} in the following proof. By the definition of soundness, we prove the following result:

$$\mathsf{SO}^{-1}(t) \subseteq T \cup T'_{\text{rd}} \tag{5.43}$$

$$\left((\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}}); \mathsf{RW}_{\mathcal{X}}{}^{?} \cup \mathsf{WW}_{\mathcal{X}}\right)^{-1}(t) \subseteq T \cup T'_{\text{rd}} \cup T''_{\text{rd}} \tag{5.44}$$

$$I_{\texttt{CP}}(\mathcal{X}', cl) \subseteq \mathsf{VisTrans}(\mathsf{XToK}(\mathcal{X}'), u') \tag{5.45}$$

Eq. (5.43) can be proven in the same way as in monotonic read on page 92. We now prove Eq. (5.44). Similar to causal consistency on page 96, initially we take T''_{rd} to be an empty set. By Theorem 5.27 and Eq. (5.28), there exists T''_{rd} such that $T \cup T''_{\text{rd}}$ is closed under $\left((\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}}); \mathsf{RW}_{\mathcal{X}}{}^{?} \cup \mathsf{WW}_{\mathcal{X}}\right)$. Now consider a transaction $t_{\text{rd}} \in T'_{\text{rd}}$ and assume a transaction t' such that $t' \xrightarrow{\left((\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}}); \mathsf{RW}_{\mathcal{X}}{}^{?} \cup \mathsf{WW}_{\mathcal{X}}\right)} t_{\text{rd}}$. Since t_{rd} is a read-only transaction, thus $t' \xrightarrow{(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}})} t_{\text{rd}}$ and the rest proof is exactly the same as as in causal consistency on page 96. Last, Eq. (5.45) can be proven in the same way as in monotonic read and read your write on pages 92 and 94 respectively. ■

The execution test $\texttt{ET}_{\texttt{CP}}$ is complete with respect to $\mathcal{A}_{\texttt{CP}}$. By Theorem 5.30, it suffice to prove that it is complete with respect to the following definition,

$$\{\lambda\mathcal{X}.\mathsf{AR}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}}, \lambda\mathcal{X}.\,\mathsf{SO}\}$$

We prove that $\texttt{ET}_{\texttt{CP}}$ is **complete** respective to $\mathcal{A}_{\texttt{CP}}$. Assume an abstract execution $\mathcal{X}$ that satisfies $\mathcal{A}_{\texttt{CP}}$ and a transaction $t \in \mathcal{X}$. Assume i^{th} and $(i+1)^{\text{th}}$ cuts, $\mathcal{X}' = (\mathcal{X}, i)$ and $\mathcal{X}'' = (\mathcal{X}, i+1)$. Let $t^n_{cl} = \mathcal{X}'' \setminus \mathcal{X}'$ be the next transaction, $T = \mathsf{VIS}^{-1}_{\mathcal{X}}(t^n_{cl})$ be the set of visible transactions, and $u = \mathsf{GetView}(\mathcal{X}, T)$ be the view with respect to T. By the definition of $\texttt{ET}_{\texttt{CP}}$, we prove $\texttt{CanCommit}_{\texttt{CP}}$, $\texttt{ViewShift}_{\texttt{MR}}$ and $\texttt{ViewShift}_{\texttt{RYW}}$ respectively. Recall that

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

Then $\texttt{CanCommit}_{\texttt{CP}}$ can be derived from Theorem 5.27 and Eq. (5.29). The predicate $\texttt{ViewShift}_{\texttt{RYW}}$ can be proven in the same way as in read your write on page 94. Because $\mathsf{VIS}_{\mathcal{X}}; \mathsf{SO} \subseteq \mathsf{AR}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$, then $\texttt{ViewShift}_{\texttt{MR}}$ can be proven in the same way as in monotonic read on page 92. ■

Parallel Snapshot Isolation An abstract execution $\mathcal{X}$ satisfies parallel snapshot isolation (PSI), if it satisfies $\{\lambda\mathcal{X}.\mathsf{VIS}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}}, \lambda\mathcal{X}.\,\mathsf{SO}, \lambda\mathcal{X}.\mathsf{WW}_{\mathcal{X}}\}$), which is intersection of CC and UA on abstract executions [Cerone et al., 2015b,a]. Similar to causal consistency on page 96, there exists a minimum visibility relation.

Theorem 5.31 (Minimum visibility relation for (PSI))**.** *For two abstract executions $\mathcal{X}, \mathcal{X}'$, the following constrain on visibility,*

$$(\mathsf{WR}_{\mathcal{X}} \cup \mathsf{WW}_{\mathcal{X}} \cup \mathsf{SO}); \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}} \qquad \mathsf{SO} \subseteq \mathsf{VIS}_{\mathcal{X}} \qquad \mathsf{WW}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}} \tag{5.46}$$

is equivalent to

$$\mathsf{VIS}_{\mathcal{X}'}; \mathsf{VIS}_{\mathcal{X}'} \subseteq \mathsf{VIS}_{\mathcal{X}'} \qquad \mathsf{SO} \subseteq \mathsf{VIS}_{\mathcal{X}'} \qquad \mathsf{WW}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}} \tag{5.47}$$

in that $\forall t \in \mathit{TxID}. \forall \mathcal{F}. (\mathcal{F} = \mathcal{T}_{\mathcal{X}}(t) \Leftrightarrow \mathcal{F} = \mathcal{X}'(t)) \wedge \mathsf{AR}_{\mathcal{X}} = \mathsf{AR}_{\mathcal{X}'}$.

Proof. The proof is similar to Theorem 5.28. For $\mathcal{X}$ that satisfies Eq. (5.46), we construct a new abstract execution $\mathcal{X}'$ with a new visibility relation $\mathsf{VIS}_{\mathcal{X}'}$ such that $\mathsf{VIS}_{\mathcal{X}'} = (\mathsf{WR}_{\mathcal{X}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{X}})^+$ and for the similar reason as Lemma 5.29, the new abstract execution $\mathcal{X}'$ satisfies Eq. (5.47). Assume an abstract execution $\mathcal{X}'$ that satisfies Eq. (5.47). Since $\mathsf{WR}_{\mathcal{X}'} \subseteq \mathsf{VIS}_{\mathcal{X}'}$ by the definition of $\mathsf{WR}_{\mathcal{X}'}$, thus $\mathcal{X}'$ satisfies Eq. (5.46). ■

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

$$\mathcal{A}_{\texttt{PSI}} \stackrel{def}{=} \{\lambda\mathcal{X}.\,(\mathsf{WR}_{\mathcal{X}} \cup \mathsf{WW}_{\mathcal{X}} \cup \mathsf{SO})\,; \mathsf{VIS}_{\mathcal{X}}, \lambda\mathcal{X}.\,\mathsf{SO}, \lambda\mathcal{X}.\,\mathsf{WW}_{\mathcal{X}}\}\,.$$

We pick the invariant as $I_{\texttt{PSI}} = I_{\texttt{MR}} \cup I_{\texttt{RYW}}$. We prove that $\texttt{ET}_{\texttt{PSI}}$ is **sound** respective to $\mathcal{A}_{\texttt{PSI}}$. Assume two kv-stores $\mathcal{K}, \mathcal{K}'$, two views u, u' and a transaction t for a client cl, a fingerprint $\mathcal{F}$ such that: (1) $(\mathcal{K}, u) \xrightarrow{\mathcal{F}}_{\texttt{PSI}} (\mathcal{K}', u')$; (2) $t \in \mathsf{NextTxID}(\mathcal{K}, cl)$; and (3) the new kv-sotre $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t)$. Assume an abstract execution $\mathcal{X}$ such that $\mathcal{K} = \mathsf{XToK}(\mathcal{X})$. Let $T = \mathsf{VisTrans}(\mathcal{K}, u)$ be the set of visible transactions and T_{rd} be a set of read-only transactions defined by:

$$T_{\mathrm{rd}} \supseteq \bigcup_{\{t^i_{cl} \mid t^i_{cl} \in \mathcal{X}\}} \left(\mathsf{VIS}^{-1}_{\mathcal{X}}(t^i_{cl}) \cup (\mathsf{SO}^{-1})^?(t^i_{cl})\right) \cap \mathsf{ReadOnlyTrans}(\mathcal{X})\,.$$

Let $\mathcal{X}' = \mathsf{UpdateAExec}(\mathcal{X}, T \cup T_{\mathrm{rd}}, \mathcal{F}, t)$ be the new abstract execution. We now check if $\mathcal{X}'$

satisfies Eqs. (5.24) and (5.25) in Def. 5.23. Assume

$$T'_{\mathrm{rd}} = \bigcup_{\{t^i_{cl} \mid t^i_{cl} \in \mathcal{X}\}} \left(\mathsf{VIS}^{-1}_{\mathcal{X}} \left(t^i_{cl}\right) \cup \left(\mathsf{SO}^{-1}\right)^{?} \left(t^i_{cl}\right) \right) \cap \mathsf{ReadOnlyTrans}\left(\mathcal{X}\right).$$

and $T''_{\mathrm{rd}} = T_{\mathrm{rd}} \setminus T'_{\mathrm{rd}}$. We construct T''_{rd} in the following proof. By the definition of soundness, we prove the following result

$$\mathsf{SO}^{-1}(t) \subseteq T \cup T'_{\mathrm{rd}} \tag{5.48}$$

$$\mathsf{WW}^{-1}(t) \subseteq T \tag{5.49}$$

$$\left(\mathsf{WR}_{\mathcal{X}'} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{X}'}\right)^{-1}(t) \subseteq T \cup T'_{\mathrm{rd}} \cup T''_{\mathrm{rd}} \tag{5.50}$$

$$I_{\mathtt{PSI}}\left(\mathcal{X}', cl\right) \subseteq \mathsf{VisTrans}\left(\mathsf{XToK}\left(\mathcal{X}'\right), u'\right) \tag{5.51}$$

Eqs. (5.48) and (5.49) can be proven in the same way as in monotonic read on page 92 and update atomic on page 99 respectively. We now prove Eq. (5.50). Initially we take T''_{rd} to be an empty set. Note that $\mathsf{VIS}^{-1}_{\mathcal{X}'}(t) = T \cup T'_{\mathrm{rd}} \cup T''_{\mathrm{rd}}$. By Theorem 5.27 and Eq. (5.28), there exists T''_{rd} such that $T \cup T''_{\mathrm{rd}}$ is closed under $\mathsf{WR}_{\mathcal{X}'} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{X}'}$. Now consider a transaction $t_{\mathrm{rd}} \in T'_{\mathrm{rd}}$ and assume a transaction t' such that $t' \xrightarrow{\mathsf{WR}_{\mathcal{X}'} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{X}'}} t_{\mathrm{rd}}$. Since t_{rd} is a read-only transaction, thus $t' \xrightarrow{(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}})} t_{\mathrm{rd}}$ and the rest proof is exactly the same as in causal consistency on page 96. Last, Eq. (5.51) can be proven in the same way as in monotonic read and read your write on pages 92 and 94 respectively. ■

The execution test $\mathtt{ET}_{\mathtt{PSI}}$ is complete with respect to the axiomatic definition $\mathcal{A}_{\mathtt{PSI}}$. By Theorem 5.31, it suffices to prove completeness with respect to the following definition,

$$\{\lambda\mathcal{X}.\, \mathsf{VIS}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}}, \lambda\mathcal{X}.\, \mathsf{SO}, \lambda\mathcal{X}.\, \mathsf{WW}_{\mathcal{X}}\}$$

We prove that $\mathtt{ET}_{\mathtt{PSI}}$ is **complete** respective to $\mathcal{A}_{\mathtt{PSI}}$. Assume an abstract execution $\mathcal{X}$ that satisfies $\mathcal{A}_{\mathtt{PSI}}$ and a transaction $t \in \mathcal{X}$. Assume i^{th} and $(i+1)^{\mathrm{th}}$ cuts, $\mathcal{X}' = (\mathcal{X}, i)$ and $\mathcal{X}'' = (\mathcal{X}, i+1)$. Let $t^n_{cl} = \mathcal{X}'' \setminus \mathcal{X}'$ be the next transaction, $T = \mathsf{VIS}^{-1}_{\mathcal{X}}\left(t^n_{cl}\right)$ be the set of visible transactions, and $u = \mathsf{GetView}\left(\mathcal{X}, T\right)$ be the view with respect to T. By the definition of $\mathtt{ET}_{\mathtt{PSI}}$, we prove $\mathtt{CanCommit}_{\mathtt{PSI}}$, $\mathtt{ViewShift}_{\mathtt{MR}}$ and $\mathtt{ViewShift}_{\mathtt{RYW}}$ respectively. Recall that $\mathtt{CanCommit}_{\mathtt{PSI}} = \mathtt{PreClosed}\left(\mathcal{K}, u, R_{\mathtt{UA}} \cup \mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}}\right)$. It is easy to see that

$$\begin{aligned} &\mathtt{PreClosed}\left(\mathcal{K}, u, R_{\mathtt{UA}} \cup \mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}}\right) \\ &\quad \Leftrightarrow \mathtt{PreClosed}\left(\mathcal{K}, u, R_{\mathtt{UA}}\right) \wedge \mathtt{PreClosed}\left(\mathcal{K}, u, \mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}}\right) \end{aligned}$$

The predicate $\mathtt{PreClosed}\left(\mathcal{K}, u, R_{\mathtt{UA}}\right)$ can be proven in the same way as in update atomic on page 99 since $\mathsf{WW}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$. Because

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

Then $\mathtt{CanCommit}_{\mathtt{CP}}$ can be derived from Theorem 5.27 and Eq. (5.29). The predicate $\mathtt{ViewShift}_{\mathtt{RYW}}$

can be proven in the same way as in read your write on page 94. Since $\mathsf{VIS}_{\mathcal{X}}; \mathsf{SO} \subseteq \mathsf{AR}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$ $\texttt{ViewShift}_{\texttt{MR}}$ can be proven in the same way as in monotonic read on page 92. ■

Snapshot Isolation An abstract execution $\mathcal{X}$ satisfies snapshot isolation ($\texttt{SI}$), if it satisfies $\{\lambda\mathcal{X}.\mathsf{AR}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}}, \lambda\mathcal{X}.\,\mathsf{SO}, \lambda\mathcal{X}.\mathsf{WW}_{\mathcal{X}}\})$, which is intersection of $\texttt{CP}$ and $\texttt{UA}$ on abstract executions [Cerone and Gotsman, 2016; Cerone et al., 2015a]. Cerone and Gotsman [2016] also proposed the minimum visibility relation that gives rise of the following equivalent definition

$$\mathcal{A}_{\texttt{SI}} \stackrel{def}{=} \left\{\lambda\mathcal{X}.\left((\mathsf{WR}_{\mathcal{X}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{X}}); \mathsf{RW}_{\mathcal{X}}{}^{?}\right); \mathsf{VIS}_{\mathcal{X}}, \lambda\mathcal{X}.\,\mathsf{SO}, \lambda\mathcal{X}.\,\mathsf{WW}_{\mathcal{X}}\right\}.$$

The execution test $\texttt{ET}_{\texttt{SI}}$ is sound with respect to the axiomatic definition $\mathcal{A}_{\texttt{SI}}$ We pick the invariant $I_{\texttt{SI}} = I_{\texttt{MR}} \cup I_{\texttt{RYW}}$. We prove that $\texttt{ET}_{\texttt{SI}}$ is **sound** respective to $\mathcal{A}_{\texttt{SI}}$. Assume two kv-stores $\mathcal{K}, \mathcal{K}'$, two views u, u' and a transaction t for a client cl, a fingerprint $\mathcal{F}$ such that: (1) $(\mathcal{K}, u) \xrightarrow{\mathcal{F}}_{\texttt{SI}} (\mathcal{K}', u')$; (2) $t \in \mathsf{NextTxID}(\mathcal{K}, cl)$; and (3) the new kv-sotre $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t)$. Assume an abstract execution $\mathcal{X}$ such that $\mathcal{K} = \mathsf{XToK}(\mathcal{X})$. Let $T = \mathsf{VisTrans}(\mathcal{K}, u)$ be the set of visible transactions and T_{rd} be a set of read-only transactions defined by:

$$T_{\text{rd}} \supseteq \bigcup_{\{t^i_{cl} \mid t^i_{cl} \in \mathcal{X}\}} \left(\mathsf{VIS}^{-1}_{\mathcal{X}}(t^i_{cl}) \cup (\mathsf{SO}^{-1})^{?}(t^i_{cl})\right) \cap \mathsf{ReadOnlyTrans}(\mathcal{X}).$$

Let $\mathcal{X}' = \mathsf{UpdateAExec}(\mathcal{X}, T \cup T_{\text{rd}}, \mathcal{F}, t)$ be the new abstract execution. We now check if $\mathcal{X}'$ satisfies Eqs. (5.24) and (5.25) in Def. 5.23. Assume

$$T'_{\text{rd}} = \bigcup_{\{t^i_{cl} \mid t^i_{cl} \in \mathcal{X}\}} \left(\mathsf{VIS}^{-1}_{\mathcal{X}}(t^i_{cl}) \cup (\mathsf{SO}^{-1})^{?}(t^i_{cl})\right) \cap \mathsf{ReadOnlyTrans}(\mathcal{X}).$$

and $T''_{\text{rd}} = T_{\text{rd}} \setminus T'_{\text{rd}}$. By the definition of soundness, we prove the following result:

$$\mathsf{SO}^{-1}(t) \subseteq T \cup T'_{\text{rd}} \tag{5.52}$$

$$\mathsf{WW}^{-1}(t) \subseteq T \tag{5.53}$$

$$\left((\mathsf{WR}_{\mathcal{X}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{X}}); \mathsf{RW}_{\mathcal{X}}{}^{?}\right)^{-1}(t) \subseteq T \cup T'_{\text{rd}} \cup T''_{\text{rd}} \tag{5.54}$$

$$I_{\texttt{PSI}}(\mathcal{X}', cl) \subseteq \mathsf{VisTrans}(\mathsf{XToK}(\mathcal{X}'), u') \tag{5.55}$$

Eqs. (5.52) and (5.53) can be proven in the same way as in monotonic read and update atomic on pages 92 and 99 respectively. We now prove Eq. (5.54). Initially we take T''_{rd} to be an empty set. Note that $\mathsf{VIS}^{-1}_{\mathcal{X}'}(t) = T \cup T'_{\text{rd}} \cup T''_{\text{rd}}$. By Theorem 5.27 and Eq. (5.28), there exists T''_{rd} such that $T \cup T''_{\text{rd}}$ is closed in the relation $\left((\mathsf{WR}_{\mathcal{X}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{X}}); \mathsf{RW}_{\mathcal{X}}{}^{?}\right)$. Now consider a transaction $t_{\text{rd}} \in T'_{\text{rd}}$ and assume a transaction t' such that $t' \xrightarrow{(\mathsf{WR}_{\mathcal{X}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{X}}); \mathsf{RW}_{\mathcal{X}}{}^{?}} t_{\text{rd}}$. Since t_{rd} is a read-only transaction, thus $t' \xrightarrow{(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{X}})} t_{\text{rd}}$ and the rest proof is exactly the same as in causal consistency on page 96. Last, Eq. (5.55) can be proven in the same way as in monotonic read and read your write on pages 92 and 94 respectively. ■

The execution test $\texttt{ET}_{\texttt{SI}}$ is complete with respect to the axiomatic definition $\mathcal{A}_{\texttt{SI}}$. By Cerone and Gotsman [2016], it suffices to prove completeness with respect to the following definition,

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

We prove that $\texttt{ET}_{\texttt{SI}}$ is **complete** respective to $\mathcal{A}_{\texttt{SI}}$. Assume an abstract execution $\mathcal{X}$ that satisfies $\mathcal{A}_{\texttt{SI}}$ and a transaction $t \in \mathcal{X}$. Assume i^{th} and $(i+1)^{\text{th}}$ cuts, $\mathcal{X}' = (\mathcal{X}, i)$ and $\mathcal{X}'' = (\mathcal{X}, i+1)$. Let $t_{cl}^n = \mathcal{X}'' \setminus \mathcal{X}'$ be the next transaction, $T = \mathsf{VIS}_{\mathcal{X}}^{-1}(t_{cl}^n)$ be the set of visible transactions, and $u = \mathsf{GetView}(\mathcal{X}, T)$ be the view with respect to T. By the definition of $\texttt{ET}_{\texttt{SI}}$, we prove $\texttt{CanCommit}_{\texttt{SI}}$, $\texttt{ViewShift}_{\texttt{MR}}$ and $\texttt{ViewShift}_{\texttt{RYW}}$ respectively. Recall that $\texttt{CanCommit}_{\texttt{SI}} = \texttt{PreClosed}\left(\mathcal{K}, u, R_{\texttt{UA}} \cup \left((\mathsf{WR}_{\mathcal{X}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{X}}); \mathsf{RW}_{\mathcal{X}}{}^{?}\right)\right)$. It is easy to see that:

$$\begin{aligned}&\texttt{PreClosed}\left(\mathcal{K}, u, R_{\texttt{UA}} \cup \left((\mathsf{WR}_{\mathcal{X}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{X}}); \mathsf{RW}_{\mathcal{X}}{}^{?}\right)\right) \Leftrightarrow \\ &\quad \texttt{PreClosed}\left(\mathcal{K}, u, R_{\texttt{UA}}\right) \wedge \texttt{PreClosed}\left(\mathcal{K}, u, \left((\mathsf{WR}_{\mathcal{X}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{X}}); \mathsf{RW}_{\mathcal{X}}{}^{?}\right)\right).\end{aligned}$$

The predicate $\texttt{PreClosed}\left(\mathcal{K}, u, R_{\texttt{UA}}\right)$ can be proven in the same way as in update atomic on page 99 Given the following result:

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

then $\texttt{PreClosed}\left(\mathcal{K}, u, \left((\mathsf{WR}_{\mathcal{X}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{X}}); \mathsf{RW}_{\mathcal{X}}{}^{?}\right)\right)$ can be derived from Eq. (5.29) in Theorem 5.27. The predicate $\texttt{ViewShift}_{\texttt{RYW}}$ can be proven in the same way as in read your write on page 94. Because $\mathsf{VIS}_{\mathcal{X}}; \mathsf{SO} \subseteq \mathsf{AR}_{\mathcal{X}}; \mathsf{VIS}_{\mathcal{X}} \subseteq \mathsf{VIS}_{\mathcal{X}}$, then $\texttt{ViewShift}_{\texttt{MR}}$ can be proven in the same way as in monotonic read on page 92. ■

Serialisability The execution test $\texttt{ET}_{\texttt{SER}}$ is sound with respect to the axiomatic definition $\mathcal{A}_{\texttt{SER}} \stackrel{def}{=} \{\lambda\mathcal{X}.\,\mathsf{AR}\}$ [Cerone et al., 2015a]. We pick the invariant as $I_{\texttt{SER}}(\mathcal{X}, cl) = \emptyset$. We prove that $\texttt{ET}_{\texttt{SER}}$ is **sound** respective to $\mathcal{A}_{\texttt{SER}}$. Assume two kv-stores $\mathcal{K}, \mathcal{K}'$, two views u, u' and a transaction t for a client cl, a fingerprint $\mathcal{F}$ such that: (1) $(\mathcal{K}, u) \xrightarrow{\mathcal{F}}_{\texttt{SER}} (\mathcal{K}', u')$; (2) $t \in \mathsf{NextTxID}(\mathcal{K}, cl)$; and (3) the new kv-sotre $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t)$. Assume an abstract execution $\mathcal{X}$ such that $\mathcal{K} = \mathsf{XToK}(\mathcal{X})$. Let $T = \mathsf{VisTrans}(\mathcal{K}, u)$ be the set of visible transactions and T_{rd} be a set of read-only transactions defined by:

$$T_{\text{rd}} = \mathsf{ReadOnlyTrans}(\mathcal{X})\,.$$

Let $\mathcal{X}' = \mathsf{UpdateAExec}(\mathcal{X}, T \cup T_{\text{rd}}, \mathcal{F}, t)$ be the new abstract execution. We now check if $\mathcal{X}'$ satisfies Eqs. (5.24) and (5.25) in Def. 5.23. It is easy to see that $\mathsf{AR}_{\mathcal{X}'}^{-1}(t)$. ■

The execution test $\texttt{ET}_{\texttt{SER}}$ is complete with respect to the axiomatic definition $\mathcal{A}_{\texttt{SER}} \stackrel{def}{=} \{\lambda\mathcal{X}.\,\mathsf{AR}\}$ [Cerone et al., 2015a]. We prove that $\texttt{ET}_{\texttt{SER}}$ is **complete** respective to $\mathcal{A}_{\texttt{SER}}$. Assume an abstract execution $\mathcal{X}$ that satisfies $\mathcal{A}_{\texttt{SER}}$ and a transaction $t \in \mathcal{X}$. Assume i^{th} and $(i+1)^{\text{th}}$ cuts, $\mathcal{X}' = (\mathcal{X}, i)$ and $\mathcal{X}'' = (\mathcal{X}, i+1)$. Let $t_{cl}^n = \mathcal{X}'' \setminus \mathcal{X}'$ be the next transaction, $T = \mathsf{VIS}_{\mathcal{X}}^{-1}(t_{cl}^n)$ be

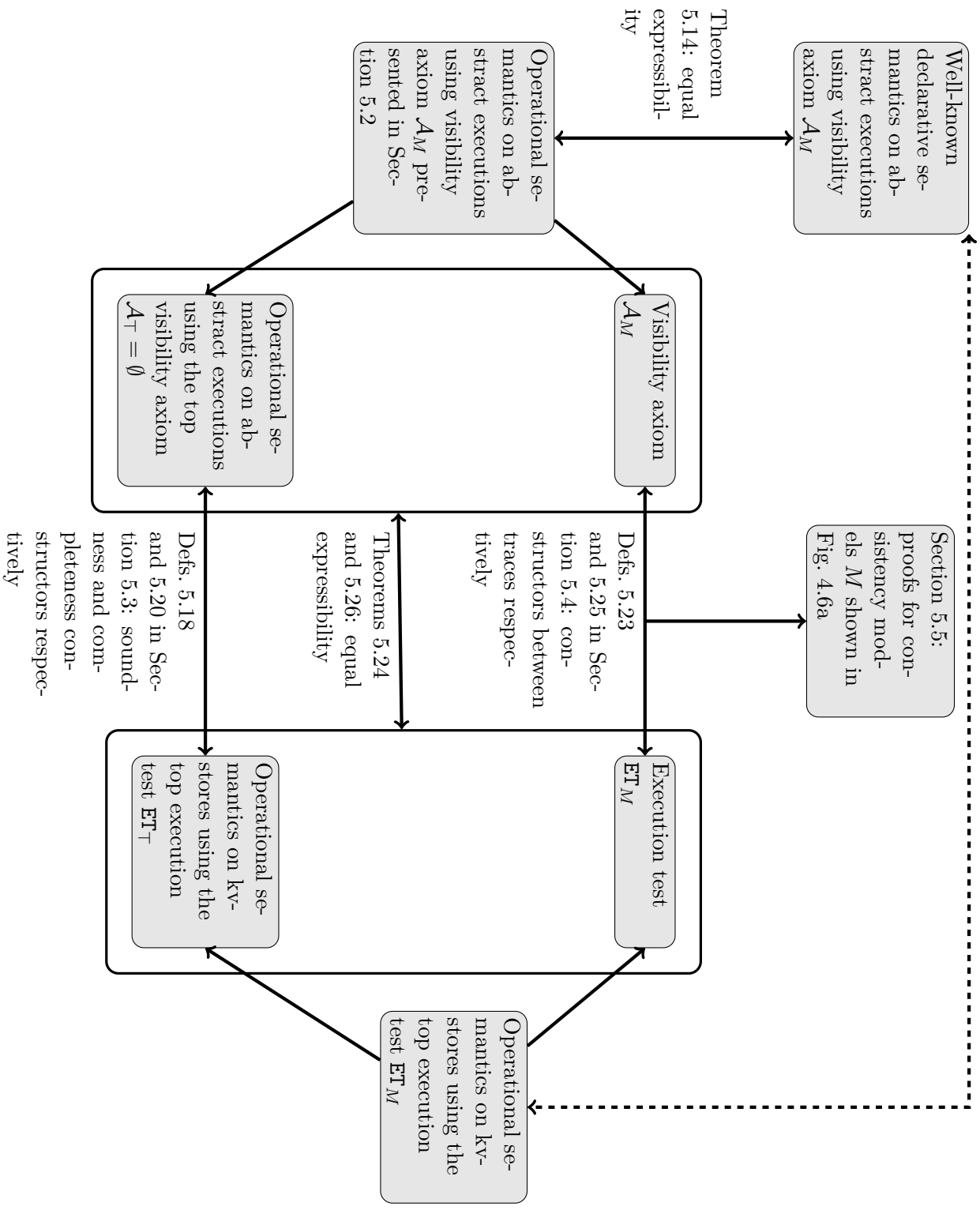


Figure 5.9: Summary of correctness proofs for definitions of consistency models M using execution tests in Fig. 4.6a

the set of visible transactions, and $u = \mathsf{GetView}(\mathcal{X}, T)$ be the view with respect to T. Since $\mathsf{AR}_\mathcal{X} = \mathsf{VIS}_\mathcal{X}$ thus it must be the case that u includes all the versions. ■

We summary the key constructors and proofs for justifying our definitions of consistency models using execution tests on kv-stores in Fig. 5.9. The goal is to prove that given a consistency model M, all the reachable states in the declarative definitions using a set of visibility axioms $\mathcal{A}_M$ on abstract execution are reachable on our operational semantics parametrised by $\texttt{ET}_M$, and vice versa (dashed line in Fig. 5.9). We first propose an operational semantics on abstract executions, which are equivalent to the axiomatic semantics (Section 5.2). This alternative semantics is also parametrised by a chosen consistency model. Both semantics on kv-stores and abstract executions can be de-composed to the base-line traces, that is, parametrised by *read atomic*, and the checks for consistency model M. In Section 5.3, we show the equivalence of the base-line traces: we prove simulation between traces, where, in each step, individual kv-store and abstract execution are proven equivalent using dependency graphs. In Section 5.4, we present the soundness and completeness constructors, showing the equivalence between individual transitions. If $\texttt{ET}_M$ and $\mathcal{A}_M$ satisfy the two constructors, then the constructors lift the equivalence to the level of traces. Last, in Section 5.5 we prove our definitions $\texttt{ET}_M$ satisfy conditions in the constructors with respect to the well-known definitions $\mathcal{A}_M$. In the following two chapters, we use our operational semantics to verify implementation protocols and reason client programs.

Chapter 6

Applications: Verification of Implementation Protocols

We give the formal definition of our operational semantics in Chapter 4, and show it is equivalent to the well-known declarative semantics based on abstract executions in Chapter 5. We verify two implementation protocols using our semantics: In this chapter, we show how to use our semantics to verify implementation protocols the COPS protocols in Section 6.1, a full replicated distributed database that satisfies causal consistency; and the Clock-SI protocol in Section 6.2, a fully partitioned distributed database that satisfies snapshot isolation.

In comparison to the declarative semantics described in Chapter 5, which describes the entered history of a program, and on which consistency models are defined by ruling out invalid executions, An operational semantics is much suitable for verifying implementation, using the standard trace refinement technique, which are well studied in, for example, verifying algorithms. Although there is no formal proof, both COPS [Lloyd et al., 2011] and Clock-SI [Du et al., 2013] developers informally argue the correctness in an operational style, showing invariants their systems. In this thesis, We formally verify these two protocols against our operational semantics in the following steps:

(1) we provide a formal model and semantics that are faithful to the pseudo-code;

(2) because implementation often executes transactions in a fine-grained way, we introduce *leftmover* and *rightmover*, steps that are allowed moving left, delaying these steps, or moving right, advancing these steps, and use these movers to normalise traces of the specific semantics, transferring to traces where transactions are interleaved;

(3) we then convert the normalised traces to kv-store traces by encoding the database states to kv-stores and client session states to views, and simulating fine-grained steps in implementation protocol by coarse-grained steps in our kv-store semantics; and

(4) last, we prove that each step in the kv-store traces satisfies the desired execution test.

6.1 Verification of COPS Protocol

As explained in Section 3.2, COPS is a fully replicated database that implements causal consistency. Each replica keeps all history versions of keys, where each version $\hat{\nu}$ has a unique and totally ordered identifier, and tracks a set of version identifiers, $\mathsf{DepSetOf}(\hat{\nu})$, that $\hat{\nu}$ depends on A replica r may not contain the most update to date version for each key, but r contains enough versions that can build a *causally consistent snapshot*, in the sense that if a Verizon $\hat{\nu}$ exists in r, then versions in $\mathsf{DepSetOf}(\hat{\nu})$ also exist in r. A client interacts with the database via a sequence of either single-write and multiple-read transactions, which are committed to the same replica in order. From the client point of view, a single-write transaction inserts a new version to the replica, and a multiple-read transaction reads a list of versions that are causally consistent in one step. Internally, transactions are executed in fine-grained way, yet all the intermediate steps are transparent to clients.

To verify COPS protocol, we first give a formal operational reference semantics for COPS protocol in Sections 6.1.1 and 6.1.2. In this semantics, we focus on the internal executions of the database, including single write, single read and synchronisation steps. A single-write transaction corresponds to a single write step in the semantics, while a multiple-read transaction is decomposed to several single read steps. Steps from different transactions and replicas are interleaved. In Section 6.1.3, we show that certain write steps are leftmover and certain read steps are rightmover; this allowed us to normalised traces in the COPS operational semantics. we show how to simulate a normalised COPS trace by a kv-store trace, and prove the kv-store trace satisfies causal consistency. The challenge is to convert the version order to WR and SO relation, and show the "dependence relation" and informal "causal consistency" notions in COPS can be refined to our formal causal consistency definition.

Notation. We pick the same letters for the concepts that have similar meaning with respect to those in our kv-store operational semantics, and annotate them with *hat*, for example, COPS version $\hat{\nu}$.

6.1.1 Machine States

In the COPS protocol, each replica tracks all the history versions, depicted in Fig. 6.1. There is no explicit transactional identifier. Instead, there is the *COPS version identifier*, which consists of the replica identifier which initially accepted the version and the local time when the version was accepted.

Definition 6.1 (COPS replica and version identifiers)**.** *The set of* COPS replica identifiers*, $\mathit{COPSRep} \ni r$, is a countably finite set that is totally ordered by a relation $<$. The set of local time, $\textsc{Time} \ni n$, is defined by $\textsc{Time} \stackrel{def}{=} \mathbb{N}$. The set of* COPS version identifiers*, $\mathit{COPSVerID} \ni id$,*

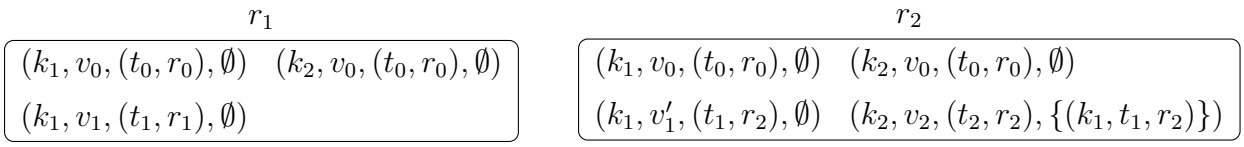


Figure 6.1: Examples of COPS states

is defined by: $\mathit{COPSVERID} \stackrel{def}{=} \mathit{TIME} \times \mathit{COPSREP}$. *The order over versions identifiers is defined by:*

$$(n, r) \sqsubseteq (n', r') \stackrel{def}{\Leftrightarrow} n < n' \vee (n = n' \wedge (r < r')$$

As explained in Section 3.2, a COPS version $\hat{\nu}$ consists of a value, a identifier and a dependency set that contains the versions on which $\hat{\nu}$ depends.

Definition 6.2 (COPS versions and dependency sets)**.** *The set of* dependency sets, $\mathit{COPSDEP} \ni d$ *is defined by:* $\mathit{COPSDEP} \stackrel{def}{=} \mathcal{P}(\mathit{KEY} \times \mathit{COPSVERID})$. *The set of* COPS versions, $\mathit{COPSVER} \ni \hat{\nu}$, *is defined by:* $\mathit{COPSVER} \stackrel{def}{=} \mathit{VALUE} \times \mathit{COPSVERID} \times \mathit{COPSDEP}$. *Let* $\mathsf{ValueOf}(\hat{\nu})$, $\mathsf{IdOf}(\hat{\nu})$ *and* $\mathsf{DepSetOf}(\hat{\nu})$ *denote the first, second and third projections of* $\hat{\nu}$ *respectively.*

A COPS database comprises a finite number of replicas. Each replica r consists of a local multi-version key-value store and a local time. A COPS replica tracks all the history versions that are uniquely identified and ordered by their writers. Without losing generality, versions for a key are organised as a list in the key-value store. However, the relative position of versions does not affect the semantics, since all versions are ordered by their timestamps.

Definition 6.3 (COPS key-value stores and databases)**.** *The set of* COPS local key-value stores *or just* COPS local stores*, is defined by:*

$$\mathit{COPSKVS} \stackrel{def}{=} \left\{ \hat{\mathcal{K}} \in \mathit{KEY} \to [\mathit{COPSVER}] \;\middle|\; \mathtt{WfCOPSKvs}\left(\hat{\mathcal{K}}\right) \right\}$$

where `WfCOPSKvs` *is defined by: for any keys* $k, k' \in \mathit{KEY}$, *indices* $i, i' \in \mathbb{N}$ *and the initial value* v_0,

$$\hat{\mathcal{K}}(k, 0) = (v_0, (r_0, 0), \emptyset), \tag{6.1}$$

$$\mathsf{IdOf}\left(\hat{\mathcal{K}}(k, i)\right) = \mathsf{IdOf}\left(\hat{\mathcal{K}}(k', i')\right) \Rightarrow k = k' \wedge i = i', \tag{6.2}$$

$$\mathsf{IdOf}\left(\hat{\mathcal{K}}(k, i)\right) \sqsubseteq \mathsf{IdOf}\left(\hat{\mathcal{K}}(k, i')\right) \Leftrightarrow i < i'. \tag{6.3}$$

Given two COPS local stores, the order, written $\hat{\mathcal{K}} \sqsubseteq \hat{\mathcal{K}}'$, *is defined by:*

$$\hat{\mathcal{K}} \sqsubseteq \hat{\mathcal{K}}' \stackrel{def}{\Leftrightarrow} \forall k \in \mathit{KEY}.\, \forall i \in \mathbb{N}.\, \forall \hat{\nu} \in \mathit{COPSVER}.\, \hat{\nu} = \hat{\mathcal{K}}(k, i) \Rightarrow \exists i' \in \mathbb{N}.\, \exists \hat{\nu}' \in \mathit{COPSVER}.\, \hat{\nu} = \hat{\mathcal{K}}'(k, i').$$

The set of COPS databases, $\mathit{COPS} \ni \mathcal{R}$, *is defined by:*

$$\mathit{COPS} \stackrel{def}{=} \left\{ \mathcal{R} \in \mathit{COPSRep} \stackrel{fin}{\rightharpoonup} (\mathit{COPSKVS} \times \mathit{Time}) \;\middle|\; \begin{array}{c} \forall r, r' \in \mathsf{Dom}(\mathcal{R}) . \forall \hat{\mathcal{K}}, \hat{\mathcal{K}}' \in \mathit{COPSKVS}. \\ \forall k, k' \in \mathit{Key}. \forall i, i' \in \mathbb{N}. \\ \mathcal{R}(r) = (\hat{\mathcal{K}}, \_) \wedge \mathcal{R}(r') = (\hat{\mathcal{K}}', \_) \wedge \\ \left(\mathsf{IdOf}\left(\hat{\mathcal{K}}(k,i)\right) = \mathsf{IdOf}\left(\hat{\mathcal{K}}'(k',i')\right) \Leftrightarrow \right. \\ \left. k = k' \wedge \hat{\mathcal{K}}(k,i) = \hat{\mathcal{K}}'(k',i') \right) \end{array} \right\}.$$

A COPS version will be eventually replicated to all replica. This means that versions with the same identifier in different replicas must be the same version. This is captured by the well-formed condition for COPS.

Each COPS client cl maintains a local context tracking versions that cl have read or written. When a client commits a new version, the client context becomes the dependency set of the new version. Hence, we model both dependency sets and client contexts as sets of pairs comprising keys and replica identifiers. Each client cl also tracks the replica identifier with which cl interacts. Note that a client always interacts with the same replica.

Definition 6.4 (COPS client contexts and environments)**.** *The set of* COPS client contexts, $\mathit{COPSCtx} \ni \hat{u}$, *is defined by* $\mathit{COPSCtx} \stackrel{def}{=} \mathcal{P}(\mathit{Key} \times \mathit{COPSVerID})$. *The set of* COPS client context environments, $\mathit{COPSCtxEnv} \ni \hat{\mathcal{U}}$, *is defined by:* $\mathit{COPSCtxEnv} \stackrel{def}{=} \mathit{CID} \stackrel{fin}{\rightharpoonup} (\mathit{COPSCtx} \times \mathit{COPSRep})$.

6.1.2 Reference Implementation and Reference Semantics

eparagraphCOPS API. A COPS client interacts with a replica via two fixed APIs: `put` and `read`. The client commits a new value v for a key k to a replica r by calling $\texttt{put}(k, v)$. Upon receiving the new value v for k, the replica assigns a new version identifier to the value, sends back the acknowledgement to the client and then broadcasts the new version to other replicas via synchronisation messages (Fig. 6.1). COPS only allows writing of one key per transaction. By contrast, a client can read a list of keys K by calling $\texttt{read}(K)$. Upon receiving the request, the replica prepares a snapshot for K in a fine-grained way, reading one key at a time. To track the intermediate states of $\texttt{read}(K)$, we introduce *fine-grained commands* COPSRuntimeCmd. As explained in Section 3.2, a replica prepares the causally dependent snapshot for K in two phases. For brevity, we encoding the runtime intermediate state in the syntax. The syntax $\texttt{read}(K) : \hat{\mathcal{V}}$ corresponds to the optimistic-read phase. The version buffer $\hat{\mathcal{V}}$ initially is empty. For each key in K, the client reads the current latest version for next key in K and puts the version in $\hat{\mathcal{V}}$. The syntax $\texttt{read}(K) : (\hat{\mathcal{V}}, D)$ corresponds to the phase change. The re-fetch set D contains, for each key in K, the version with the maximum version identifier over versions $\hat{\nu}$ in $\hat{\mathcal{V}}$ or any versions on which $\hat{\nu}$ depend. The syntax $\texttt{read}(K) : (\hat{\mathcal{V}}, D, \hat{\mathcal{V}}')$ corresponds to the re-fetch phase. For each key in K, the client reads the version ν contained in D and put it

in $\hat{\mathcal{V}}'$, if ν has greater version identifier than the version in $\hat{\mathcal{V}}$. Otherwise, the client copies the version in $\hat{\mathcal{V}}$ to $\hat{\mathcal{V}}'$.

Definition 6.5 (COPS commands and programs)**.** *The set of* COPS commands*,* $\mathit{COPSCmd} \ni \hat{\mathtt{C}}$*, is defined by:*

$$\hat{\mathtt{C}} ::= \mathtt{put}\,(k, v)\,|\mathtt{read}\,(K)\,|\hat{\mathtt{C}}\,;\hat{\mathtt{C}}$$

for any key $k \in \mathit{Key}$*, key set* $K \subseteq \mathit{Key}$ *and* $v \in \mathit{Value}$*. The set of* COPS fine-grained commands*,* $\mathit{COPSRuntimeCmd} \ni \mathtt{R}$*, is defined by:*

$$\mathtt{R} ::= \mathtt{put}\,(k, v)\,|\mathtt{read}\,(K)\,|\mathtt{read}\,(K) : \hat{\mathcal{V}}|\mathtt{read}\,(K) : (\hat{\mathcal{V}}, D)|\mathtt{read}\,(K) : (\hat{\mathcal{V}}, D, \hat{\mathcal{V}}')|\mathtt{R}\,;\mathtt{R}$$

for any $\hat{\mathcal{V}}, \hat{\mathcal{V}}' \in [\mathit{COPSVer}]$ *and* $D \in [\mathit{COPSDep}]$*. The sets of* COPS programs*,* $\mathit{COPSProg} \ni \hat{\mathtt{P}}$*, and* COPS fine-grained programs*,* $\mathit{COPSRuntimeProg} \ni \mathtt{I}$*, are defined by:* $\mathit{COPSProg} \stackrel{def}{=} \mathit{CID} \xrightarrow{fin} \mathit{COPSCmd}$ *and* $\mathit{COPSRuntimeProg} \stackrel{def}{=} \mathit{CID} \xrightarrow{fin} \mathit{COPSRuntimeCmd}$ *respectively.*

We now explain the reference implementation of the COPS protocol, and the semantics of the protocols. For manipulating COPS program traces, we label each transition in the traces. Given a client cl and replica r, the labels can be introduced intuitively as following:

(1) $(cl, r, (\mathtt{W}, k, v), id, d)$ denotes that cl commits a new version of a key k to a replica r with a value v, a version identifier $id = (r, n)$ and a dependency set d;

(2) $(cl, r, \mathtt{S})$ denotes that cl starts of a multi-read transaction and $\mathtt{S}$ means the start of a multiple read transaction;

(3) $(cl, r, (\mathtt{R}, k, v), id, d, \mathtt{Opt})$ denotes that cl reads a version of key k from r indexed by a version identifier id with a value v and a dependency set d in the optimistic-read phase, and $\mathtt{Opt}$ means a optimistic read;

(4) $(cl, r, \mathtt{P})$ denotes that cl starts re-fetch phase and $\mathtt{P}$ means phase change;

(5) $(cl, r, (\mathtt{R}, k, v), id, d, \mathtt{Ref})$ denotes that cl reads a version of key k from r indexed by a version identifier id with a value v and a dependency set d in the re-fetch phase, and $\mathtt{Ref}$ means a re-fetch read;

(6) $(cl, r, \hat{u}, \mathtt{E})$ denotes that r returns the set of versions $\hat{u}$ to cl and $\mathtt{E}$ means the end of a multiple read transaction; and

(7) (r, id) denotes that r receives a synchronisation message containing a version with a identifier id.

Definition 6.6 (COPS labels)**.** *The set of* COPS labels*,* $\mathit{COPSLabels} \ni \iota$*, is defined by:*

$$\mathit{COPSLabels} \stackrel{def}{=} \bigcup_{\substack{cl \in \mathit{CID}, r \in \mathit{COPSRep} \\ k \in \mathit{Key}, v \in \mathit{Value} \\ id \in \mathit{COPSVerID}, d \in \mathit{COPSDep}}} \left\{ \begin{array}{c} (cl, r, \mathtt{S}), (cl, r, (\mathtt{W}, k, v), id, d), \\ (cl, r, (\mathtt{R}, k, v), id, d, \mathtt{Opt}), (cl, r, \mathtt{P}), \\ (cl, r, (\mathtt{R}, k, v), id, d, \mathtt{Ref}), (cl, r, \hat{u}, \mathtt{E}), (r, id) \end{array} \right\}.$$

COPS provides two APIs: `put` and `read`. A client will send the request to a replica and awaiting a reply. For these two APIs, we explain the reference implementation first and give the semantics. We argue that the semantics captures the implementation.

Reference implementation for `put`. When a client calls `put` function depicted in Fig. 6.2a, it sends a new value `v` for key `k` with the context `ctx` to a replica `repl`. Upon receiving the new value, replica r creates a new version for `k` where:

(1) the value is `v`;

(2) the identifier consists of the next available local time-stamp `ltime` (line 3) and the replica identifier `repl.id` (line 5); and

(3) the dependency set is the client context `deps` = `ctx` (line 4).

Replica `repl` then commits the new version to its local store via `list_insert` (line 6), which inserts the new version `(v, id, deps)` to the list `repl.kv[k]` in the sense that the new list preserves the order over version identifiers. Now the replica is ready to send back the acknowledgement message to the client comprising the new version identifier. Upon receiving the acknowledgement message, the client updates the client context, adding this new version in its context (line 8). Last, replica `repl` broadcasts this new version by calling `asyn_broadcast`.

Reference semantics for `put`. The semantics of `put` is captured by rule COPSWrite in Fig. 6.2b. The configuration comprises a COPS store $\hat{\mathcal{K}}$, a client context $\hat{u}$ and a replica local time n. The first and second premises construct the new identifier $id = (r, n+1)$ and the new version $\hat{\nu}$ as expected. This new version is inserted into $\hat{\mathcal{K}}$ via $\mathsf{COPSInsert}$, updating the COPS store to $\hat{\mathcal{K}}' = \mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k, \hat{\nu}\right)$.

Definition 6.7 (List insertion)**.** *Given a COPS store* $\hat{\mathcal{K}} \in \mathit{COPSKVS}$ *and a version* $\hat{\nu} \in \mathit{COPSVer}$ *for a key* $k \in \mathit{Key}$*, the function* $\mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k, \hat{\nu}\right)$ *is defined by:*

$$\begin{array}{l} \mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k, \hat{\nu}\right) \stackrel{def}{=} \mathtt{let}\ \hat{\mathcal{V}} = \hat{\mathcal{K}}(k)\ \mathtt{and}\ [\hat{\nu}_0, \cdots, \hat{\nu}_i, \hat{\nu}_{i+1}, \cdots, \hat{\nu}_n] = \hat{\mathcal{V}} \\ \qquad\qquad \mathtt{where}\ \mathsf{IdOf}(\hat{\nu}_i) \sqsubseteq \mathsf{IdOf}(\hat{\nu}) \sqsubseteq \mathsf{IdOf}(\hat{\nu}_{i+1}) \\ \qquad\qquad\qquad \mathtt{in}\ \hat{\mathcal{K}}\left[k \mapsto [\hat{\nu}_0, \cdots, \hat{\nu}_i, \hat{\nu}, \hat{\nu}_{i+1}, \cdots, \hat{\nu}_n]\right] \end{array}$$

```
1 // mixing the client API and system API
2 'repl' receive 'put(k,v)' request from a client with 'ctx' {
3     ltime = inc(repl.local_time);
4     deps = ctx;
5     txid = (ltime ++ repl.id);
6     list_insert(repl.kv[k],(v, txid, deps));
7     return txid;
8     ctx += (k, txid);
9     asyn_broadcast(k, v, txid, deps);
10 }
```

(a) Reference implementation for `put`

COPSWRITE

$$\frac{\#\mathbf{3},\mathbf{5} : id = (r, n+1) \qquad \#\mathbf{6} : \hat{\nu} = (v, id, \hat{u}) \qquad \#\mathbf{6} : \hat{\mathcal{K}}' = \mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k, \hat{\nu}\right) \qquad \#\mathbf{8} : \hat{u}' = \hat{u} \uplus \{(k, id)\}}{\left(\hat{\mathcal{K}}, \hat{u}, n\right), \mathtt{put}\ (k, v) \xrightarrow{(cl,r,(\mathtt{W},k,v),id,\hat{u})} \left(\hat{\mathcal{K}}', \hat{u}', n+1\right), \mathtt{skip}}$$

(b) Reference semantics for `put`, where #**n** denotes line n in Fig. 6.2a

$$k_1 \mapsto \boxed{v_0 \mid id_0 \mid d_0}\boxed{v_2 \mid id_2 \mid d_2} \xrightarrow{\mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k, (v_1, id_1, d_1)\right)} k_1 \mapsto \boxed{v_0 \mid id_0 \mid d_0}\boxed{v_1 \mid id_1 \mid d_1}\boxed{v_2 \mid id_2 \mid d_2}$$

(c) An example of COPSInsert, where $id_0 \sqsubseteq id_1 \sqsubseteq id_2$

Figure 6.2: COPS API: `put`

where the version identifier order $\sqsubseteq$ is defined in Def. 6.1.

The function $\hat{\mathcal{K}}' = \mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k, \hat{\nu}\right)$ inserts the version $\hat{\nu}$ in a position that preserves the order, depicted in Fig. 6.2c. This resulting $\hat{\mathcal{K}}'$ is well-formed (the proof is given in appendix A.6 on page 204). The last premise in rule COPSWRITE in Fig. 6.2b, $\hat{u}' = \hat{u} \uplus \{(k, id)\}$, updates the client context, incorporating the new version in the new context $\hat{u}'$.

Reference implementation for `read`. Fig. 6.3a presents the reference implementation for `read`, where a client requests a snapshot for a list of distinct keys `ks` from a replica r. As explained in Section 3.2, the replica constructs this snapshot in two phases. In the first optimistic-read phase (line 3 in Fig. 6.3a), the client reads the latest version for each key using `get_by_version` with `LATEST` label, one key at a time. However, interleaving may happen between two reads.

At the end of the first phase, the client collects the versions (line 4) taken in the optimistic phase and their dependency sets (line 7), and computes a *re-fetch set*, which determines versions that will be fetched in the second phase. Specifically, the re-fetch set, `ccv` in line 7, contains the version with the maximum time-stamp for each key. The client initialises `ccv` with versions

fetched in the optimistic phase, and traverses all dependency entry `dep` in all dependency sets `deps`. If `dep` contains a version of a key `dep.key` in `ks`, and if the version identifier `dep.id` is greater than the current version identifier stored in the `ccv[dep.key]`, then `ccv[dep.key]` is updated to `dep.id`.

In the re-fetch phase, for each key `k` in `ks`, if the version fetched in the first phase, `rst[k].vers`, has smaller time-stamp than that of the version in the re-fetch set, `ccv`[`k`], then client should re-fetch the version of `ccv`[`k`] (line 10). At the end of this re-fetch phase, `rst` contains the snapshot, which Lloyd et al. [2011] informally argued that the snapshot should guarantee causal consistency in the sense that any reads from the snapshot must satisfy the closure of visibility. This aligns with the axiom of $\mathsf{VIS}; \mathsf{VIS} \subseteq \mathsf{VIS}$ on abstract executions. Last, this snapshot, `rst`, is sent back to the client, and the client adds all the versions in `rst` to the context (line 11).

Reference semantics for `read`. The semantics for `read` are given in Fig. 6.3b. The rule COPSSTART (Fig. 6.3b) checks the uniqueness of K by predicate $\texttt{Unique}\,(K)$ and initialises the first phase, updating the command to $\texttt{read}\,(K) : \emptyset$ with an empty list $\emptyset$. This transition does not access the store, hence is labelled with $(cl, r, \texttt{S})$. This optimistic-read phase is captured by rule COPSOPT in Fig. 6.3b. Client cl reads the latest version $\hat{\nu}_i$ in $\hat{\mathcal{K}}$ for next key k_i: that is, $\hat{\nu}_i = \hat{\mathcal{K}}\left(k, \left|\hat{\mathcal{K}}\,(k)\right| - 1\right)$. The resulting fine-grained command tracks version $\hat{\nu}_i$ in the version list $(\hat{\mathcal{V}} :: [\hat{\nu}_i])$, analogous to `rst` in line 3. This transition is labelled with the information including client cl, replica r, read operation $(\texttt{R}, k_i, v_i)$, the version identifier id_i and the dependency set d_i, The `Opt` in the label indicates the first phase.

The phase change is captured by rules COPSSSET, which computes the re-fetch set, analogous to lines 4 and 7. The function $D = \mathsf{VerLower}\left(K, \hat{\mathcal{V}}\right)$ is the re-fetch set of K, in the sense that $D_{|i}$ is the version of $K_{|i}$, which has the maximum time-stamp over versions of $K_{|i}$ that is either $\hat{\mathcal{V}}_{|i}$ or any versions on which $\hat{\mathcal{V}}$ depends.

Definition 6.8 (Re-fetch set)**.** *Given a key set K and list of versions for the key set $\hat{\mathcal{V}}$ the function* $\mathsf{VerLower}\left(K, \hat{\mathcal{V}}\right)$ *is defined by:*

$$\begin{aligned}
\mathsf{VerLower}\left(K, \hat{\mathcal{V}}\right) &\stackrel{def}{=} \texttt{let}\ n = |K| - 1\ \texttt{and}\ \ [k_0, \cdots, k_n] = K \\
&\qquad\qquad \texttt{in}\ \left[\mathsf{VerLowerN}\left(k_0, \hat{\mathcal{V}}, 0\right), \cdots, \mathsf{VerLowerN}\left(k_n, \hat{\mathcal{V}}, n\right)\right] \\
\mathsf{VerLowerN}\left(k, \hat{\mathcal{V}}, i\right) &\stackrel{def}{=} \mathsf{Max}_{\sqsubseteq}\left(\left\{id \;\middle|\; id = \mathsf{IdOf}\left(\hat{\mathcal{V}}_{|i}\right) \vee \exists i'.\, (k, id) \in \mathsf{DepSetOf}\left(\hat{\mathcal{V}}_{|i'}\right)\right\}\right).
\end{aligned}$$

The rule COPSSSET then update the fine-grained command to $\texttt{read}\,(K) : (\hat{\mathcal{V}}, D)$, tracking D. This rule is labelled with $(cl, r, \texttt{P})$, which contains the client cl, the relica r and the phase change label `P`.

The rule COPSSREFETCH in Fig. 6.3b captures the semantics of the re-fetch phase. In this phase, the client re-fetches the newer versions contained in the re-fetch set, one at a time. If the

```
1 'repl' receive 'read(ks)' request from a client with 'ctx' {
2     for k in ks
3         { rst[k] = get_by_version(repl,k,LATEST); }
4     for k in ks { ccv[k] = rst[k].id; }
5     for k in ks {
6         for dep in rst[k].deps
7             if ( dep.key ∈ ks ) ccv[dep.key] = max (ccv[dep.key], dep.id);
8     }
9     for k in ks
10        if ( ccv[k] > rst[k].id ) rst[k] = get_by_version(repl,k,ccv[k]);
11    for (k,id,deps) in rst { ctx += (k,id,deps); }
12    return to_vals(rst);
13 }
```

(a) Reference implementation for `read`

COPSSTART

$$\frac{\texttt{Unique}\,(K)}{\left(\hat{\mathcal{K}}, \hat{u}, n\right), \texttt{read}\,(K) \xrightarrow{(cl,r,\mathtt{S})} \left(\hat{\mathcal{K}}, \hat{u}, n\right), \texttt{read}\,(K) : \emptyset}$$

COPSOPT

$$\frac{\begin{array}{c}\#\mathbf{2} : K = [k_0, \cdots, k_i, \cdots, k_m] \qquad \#\mathbf{2} : \hat{\mathcal{V}} = [\hat{\nu}_0, \cdots, \hat{\nu}_{i-1}] \qquad \#\mathbf{2} : i - 1 < m \\ \#\mathbf{3} : \hat{\mathcal{K}}\left(k_i, \left|\hat{\mathcal{K}}\,(k_i)\right| - 1\right) = (v_i, id_i, d_i) = \hat{\nu}_i \qquad \iota = (cl, r, (\mathtt{R}, k_i, v_i), id_i, d_i, \mathtt{Opt})\end{array}}{\left(\hat{\mathcal{K}}, \hat{u}, n\right), \texttt{read}\,(K) : \hat{\mathcal{V}} \xrightarrow{\iota} \left(\hat{\mathcal{K}}, \hat{u}, n\right), \texttt{read}\,(K) : (\hat{\mathcal{V}} :: [\hat{\nu}_i])}$$

COPSSET

$$\frac{|K| = \left|\hat{\mathcal{V}}\right| \qquad \#\mathbf{7} : D = \mathsf{VerLower}\left(K, \hat{\mathcal{V}}\right)}{\left(\hat{\mathcal{K}}, \hat{u}, n\right), \texttt{read}\,(K) : \hat{\mathcal{V}} \xrightarrow{(cl,r,\mathtt{P})} \left(\hat{\mathcal{K}}, \hat{u}, n\right), \texttt{read}\,(K) : (\hat{\mathcal{V}}, D)}$$

COPSREFETCH

$$\frac{\begin{array}{c}\#\mathbf{5} : K = [k_0, \cdots, k_i, \cdots, k_m] \\ \hat{\mathcal{V}} = [\hat{\nu}_0, \cdots, \hat{\nu}_m] \qquad \#\mathbf{5} : \hat{\mathcal{V}}' = \left[\hat{\nu}'_0, \cdots, \hat{\nu}'_{i-1}\right] \qquad \#\mathbf{5} : i - 1 < m \\ D_{|i} = id \qquad \#\mathbf{10} : (v_i, id_i, d_i) = \begin{cases} \hat{\mathcal{K}}\,(k_i, j) & \text{if } \mathsf{IdOf}\,(\hat{\nu}_i) \sqsubseteq id \wedge id = \mathsf{IdOf}\left(\hat{\mathcal{K}}\,(k_i, j)\right) \\ \hat{\nu}_i & \text{otherwise} \end{cases} \\ \#\mathbf{10} : \hat{\mathcal{V}}'' = \hat{\mathcal{V}}' :: [(v_i, id_i, d_i)] \qquad \iota' = (cl, r, (\mathtt{R}, k_i, v_i), id_i, d_i, \mathtt{Ref})\end{array}}{\left(\hat{\mathcal{K}}, \hat{u}, n\right), \texttt{read}\,(K) : (\hat{\mathcal{V}}, D, \hat{\mathcal{V}}') \xrightarrow{\iota} \left(\hat{\mathcal{K}}, \hat{u}, n\right), \texttt{read}\,(K) : (\hat{\mathcal{V}}, D, \hat{\mathcal{V}}'')}$$

COPSFINISH

$$\frac{|K| = \left|\hat{\mathcal{V}}'\right| \qquad \#\mathbf{11} : \hat{u}' = \left\{(k, id) \;\middle|\; \exists i.\, k = K_{|i} \wedge (\_, id, \_) = \hat{\mathcal{V}}'_{|i}\right\}}{\left(\hat{\mathcal{K}}, \hat{u}, n\right), \texttt{read}\,(K) : (\hat{\mathcal{V}}, D, \hat{\mathcal{V}}') \xrightarrow{(cl,r,\hat{u}',\mathtt{E})} \left(\hat{\mathcal{K}}, \hat{u} \cup \hat{u}', n\right), \texttt{skip}}$$

(b) Reference semantics for `read`, where #**n** denotes line n in Fig. 6.3a

Figure 6.3: COPS API: `read`

version identifier contained in the re-fetch set, $id = D_{|i}$, for the key ,k_i, has greater time-stamp than the version of k_i read in the first phase, that is, $\hat{\nu}_i$, then it re-fetches the version identified by id, that is, $\hat{\mathcal{K}}\,(k_i, j)$. Otherwise, there is no need to re-fetch for the key k_i. In the resulting buffer $\mathcal{V}''$, we track either the re-fetched version of k_i or copy the same version, $\hat{\nu}_i$, from the first

phase. Note that versions in $\mathcal{V}''$ matches those in `rst` in line 10. The label of this transition tracks the client cl, the replica r, the key k_i, the value v_i, the version identifier id_i and the dependency set d_i. The label `Ref` denotes re-fetch phase. At the end of the re-fetch phase, the final snapshot is returned to the client, which is subsequently added in the context. The rule COPSFinish captures line 11 in the reference implementation. This rule update the client context to $\hat{u} \cup \hat{u}'$, and the command to `skip`.

The rules for read and write are lifted to a program-level interleaving semantics presented in Fig. 6.4b. The rule COPSClient captures that a client takes a fine-grained step. A COPS configuration Θ comprises a COPS database $\mathcal{R}$ and a COPS client environment $\hat{\mathcal{U}}$. Any client from the environment $\hat{\mathcal{U}}$ must be served by a replica in $\mathcal{R}$.

Definition 6.9 (COPS configurations)**.** *The set of* COPS configurations*, $COPSConf \ni \Theta$, is defined by:*

$$COPSConf \stackrel{def}{=} \left\{ (\mathcal{R}, \hat{\mathcal{U}}) \in COPS \times COPSCtxEnv \;\middle|\; \begin{array}{l} \forall cl \in \mathsf{Dom}\left(\hat{\mathcal{U}}\right). \forall r \in COPSRep. \\ \quad \hat{\mathcal{U}}(cl) = (\_, r) \Rightarrow r \in \mathsf{Dom}(\mathcal{R}) \end{array} \right\}.$$

The set of initial COPS configurations*, $COPSConf_0 \ni \Theta_0$, is defined by:*

$$COPSConf_0 \stackrel{def}{=} \left\{ (\mathcal{R}, \hat{\mathcal{U}}) \in COPSConf \;\middle|\; \begin{array}{l} \forall r \in COPSRep. \forall cl \in CID. \\ \quad \mathcal{R}(r) = ((\lambda k \in Key. [(v_0, t_0, \emptyset)]), 0) \wedge \hat{\mathcal{U}}(cl) = (\emptyset, \_) \end{array} \right\}.$$

Synchronisation Between Replicas Replicas broadcast synchronisation messages for every new version. Since versions are ordered and messages are assumed to be eventually delivered, this guarantees COPS satisfies eventual consistency: that is, all replicas eventually reach the same state. Once a replica receives a message with a new version $\hat{\nu}$, it accepts the version only if all the versions that $\hat{\nu}$ depends on exist in the replica (Fig. 6.4a). Otherwise, the replica waits for this condition to be satisfied (line 2). Last, if the message contains a later local time n in the version identifier $id = (n, r)$, the receiver replica will update its local time to n. Rule COPSSync in Fig. 6.4b models the synchronisation. Note that a COPS trace has no unfinished read operation. Two COPS traces are equivalent if and only if the last configurations match.

Definition 6.10 (COPS traces)**.** *The set of* COPS traces*, $COPSTrace \ni \zeta$, is defined by: $COPSTrace \stackrel{def}{=} \left\{ \zeta \mid \exists n, \Theta, \hat{\mathsf{P}}. \zeta \in \mathsf{COPSTraceN}(n) \wedge \mathsf{Last}(\zeta) = (\Theta, \hat{\mathsf{P}}) \wedge \hat{\mathsf{P}} \in COPSProg \right\}$, where $COPSProg$ is defined in Def. 6.5. The function* $\mathsf{COPSTraceN}$ *is defined by:*

$$\mathsf{COPSTraceN}(0) \stackrel{def}{=} \left\{ \Theta_0, \hat{\mathsf{P}}_0 \mid \Theta_0 \in COPSConf_0 \wedge \hat{\mathsf{P}}_0 \in COPSProg \right\}$$

$$\mathsf{COPSTraceN}(n+1) \stackrel{def}{=} \left\{ \zeta \xrightarrow{\iota} \Theta, \mathtt{I} \;\middle|\; \begin{array}{r} \zeta \in \mathsf{COPSTraceN}(n) \wedge \Theta \in COPSConf \\ \wedge\, \mathtt{I} \in COPSRuntimeProg \end{array} \right\}$$

```
1 'repl' receive synchronisation 'message(k,v,id,deps)' {
2     for (k',id') in deps { wait until (_,id',_) ∈ repl.kv(k'); }
3
4     list_isnert(repl.kv[k],(v,id,deps));
5     (remote_local_time ++ replid) = id;
6     repl.local_time = max(remote_local_time, repl.local_time);
7 }
```

(a) The reference implementation for COPS synchronisation

$$\rightarrow : \textsc{CopsConf} \times \textsc{CopsRuntimeProg} \times \textsc{CopsLabels} \times \textsc{CopsConf} \times \textsc{CopsRuntimeProg}$$

COPSClient

$$\frac{\begin{array}{c}\mathcal{R}(r) = (\hat{\mathcal{K}}, n) \\ \hat{\mathcal{U}}(cl) = (\hat{u}, r) \qquad \mathtt{I}(cl) = \mathtt{R} \qquad \iota = (cl, r, \cdots) \qquad \left(\hat{\mathcal{K}}, \hat{u}, n\right), \mathtt{R} \xrightarrow{\iota} \left(\hat{\mathcal{K}}', \hat{u}', n'\right), \mathtt{R}' \\ \mathcal{R}' = \mathcal{R}\left[r \mapsto (\hat{\mathcal{K}}', n')\right] \qquad \hat{\mathcal{U}}' = \hat{\mathcal{U}}\left[cl \mapsto (\hat{u}', r)\right] \qquad \mathtt{I}' = \mathtt{I}\left[cl \mapsto \mathtt{R}'\right]\end{array}}{\left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I} \xrightarrow{\iota} \left(\mathcal{R}', \hat{\mathcal{U}}'\right), \mathtt{I}'}$$

COPSSync

$$\frac{\begin{array}{c}r \neq r' \qquad \mathcal{R}(r) = (\hat{\mathcal{K}}, n) \qquad \mathcal{R}(r') = (\hat{\mathcal{K}}', n') \qquad id \notin \hat{\mathcal{K}}' \qquad \hat{\mathcal{K}}(k, i) = \hat{\nu} \qquad \mathsf{IdOf}(\hat{\nu}) = id \\ \#\mathbf{2} : \forall id'.\, (\_, id') \in \mathsf{DepSetOf}(\hat{\nu}) \Rightarrow id' \in \hat{\mathcal{K}}' \qquad \#\mathbf{4} : \hat{\mathcal{K}}'' = \mathsf{COPSInsert}\left(\hat{\mathcal{K}}', k, \hat{\nu}\right) \\ (n'', \_) = \mathsf{IdOf}(\hat{\nu}) \qquad \#\mathbf{4}, \#\mathbf{6} : \mathcal{R}\left[r' \mapsto (\hat{\mathcal{K}}'', \mathsf{Max}(n', n''))\right]\end{array}}{\left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I} \xrightarrow{(r', id)} \left(\mathcal{R}', \hat{\mathcal{U}}\right), \mathtt{I}}$$

(b) Reference semantics for COPS synchronisation and programs, where #**n** denotes line n in Fig. 6.4a

Figure 6.4: COPS synchronisation and programs

where $\textsc{CopsConf}_0$ *and* $\textsc{CopsConf}$ *are defined in Def. 6.9, and* $\textsc{CopsRuntimeProg}$ *is defined in Def. 6.5. Two traces* ζ, ζ' *are equivalent, written* $\zeta \simeq \zeta'$*, if and only if, the last configurations are equal,* $\zeta \simeq \zeta' \overset{def}{\Leftrightarrow} \mathsf{Last}(\zeta) = \mathsf{Last}(\zeta')$.

6.1.3 Verification: Annotated Normalised Traces

We introduce *normalised COPS traces*, where the re-fetch phase of any multiple read transaction is executed atomically, and *annotated normalised COPS traces*, where transactions are assigned with unique *COPS transaction identifiers*.

A *normalised COPS trace* is a trace where: (1) the order of single-write transactions in the trace agrees on the order over the transaction identifiers (Eq. (6.4)); (2) the re-fetch phase of multiple-read transactions cannot be interleaved (Eq. (6.5)). The first property ensures that any version of a key is always the latest version of the key. The second property ensures that multiple-read transactions appear to be atomic with respect to clients. In COPS, there is

no direct information about read-only transactions. We give explicit identifiers for read-only transactions in a normalised trace. We call the resulting trace an *annotated normalised COPS trace*.

Definition 6.11 (Normalised COPS traces)**.** *A* normalised COPS trace $\zeta \in$ *COPSTrace, written* $\texttt{NCOPSTrace}\,(\zeta)$*, if and only if: for the length of the trace* $n = |\zeta|$*, states* $\Theta_0, \cdots, \Theta_{n-1}$*, labels* $\iota_1, \cdots, \iota_{n-1}$*, clients* cl, cl'*, replicas* r, r'*, keys* k, k' *values* v, v'*, version identifiers* id, id'*, contexts* $\hat{u}, \hat{u}'$ *and indices* i, i'*,*

$$\begin{aligned}
&\zeta = \Theta_0 \xrightarrow{\iota_1} \cdots \xrightarrow{\iota_{n-1}} \Theta_{n-1} \\
&\quad \wedge \iota_i = (cl, r, (\texttt{W}, k, v), id, d) \wedge \iota_{i'} = (cl', r', (\texttt{W}, k', v'), id', d') \wedge i < i' \Rightarrow id \sqsubseteq id', \qquad (6.4)\\
&(\iota_i = (cl, r, \texttt{P}) \vee \iota_i = (cl, r, (\texttt{R}, k, v), id, d, \texttt{Ref})) \wedge i' = i + 1 \\
&\qquad \Rightarrow \iota_{i'} = (cl, r, (\texttt{R}, k', v'), id', d', \texttt{Ref}) \vee \iota_{i'} = (cl, r, \hat{u}, \texttt{E})\,. \qquad (6.5)
\end{aligned}$$

The normalised COPS trace is the key for proving that COPS satisfies causal consistency against our semantics. In COPSWrite rule, the new version is inserted into the list via COPSInsert that preserves the order of the new list. However, in a normalised race, by Eq. (6.4), it is enough to append the new version to the end of the list and guarantees to preserve the order (Prop. A.18). The second property, Eq. (6.5), ensures that any multiple-read transaction can be treated as an atomic step, stating that, if a step ι_i is a re-fetch set step $\texttt{P}$ or a re-fetch read step $\texttt{Ref}$ from a client cl, then the next step is a re-fetch read or a read return step $\texttt{E}$ from cl. Although the read return step and fe-fetch set step have no effect to the database, they are bundled with all re-fetch reads. These two steps are used later, as a explicit boundary for the entire re-fetch phase. Recall that in COPSRefetch rule, only the versions that are put in the buffer $\mathcal{V}''$ will be included in the snapshot to the client. This means that the atomicity of the re-fetch phase is enough for showing atomicity of the entire transaction. Every COPS trace can be transferred to its equivalent normalised COPS trace by swapping steps.

Theorem 6.12 (Equivalent normalised COPS traces)**.** *Given a COPS trace* $\zeta \in$ *COPSTrace, there is an equivalent normalised COPS trace* ζ_1 *such that* $\zeta \simeq \zeta_1 \wedge \texttt{NCOPSTrace}\,(\zeta_1)$*.*

Proof. let $\zeta_1 = \zeta$ initially. We perform the following equivalent transformation until ζ_1 is a normalised COPS trace. First, we move every re-fetch operation (in the second phase) to the right, delaying the re-fetch, until the operation is directly followed by the end of the transaction or other re-fetch operation from the same transaction. Second, we move the out-of-order write operation to the left. In the rest of the proof, we use _ to denote a machine state.

(1) **Right mover: re-fetch operation**. Take the first end of transaction step, $(cl, r, \hat{u}, \texttt{E})$, where one of its re-fetch operations, $(cl, r, (\texttt{R}, k, v), id, d', \texttt{Ref})$, is interleaved by other client: that is, for three trace segments $\zeta', \zeta'', \zeta'''$ and a step ι,

$$\zeta_1 = \zeta' \xrightarrow{(cl,r,(\texttt{R},k,v),id,d',\texttt{Ref})} \_ \xrightarrow{\iota} \zeta'' \xrightarrow{(cl,r,\hat{u},\texttt{E})} \zeta''' \wedge \iota \neq (cl, r, (\texttt{R}, k', v'), id', d'', \texttt{Ref})\,.$$

The ι can be moved left:

$$\left(\zeta' \xrightarrow{\iota} \_ \xrightarrow{(cl,r,(\mathtt{R},k,v),id,d',\mathtt{Ref})} \zeta'' \xrightarrow{(cl,r,\hat{u},\mathtt{E})} \zeta'''\right) \simeq \zeta_1.$$

If the label ι is not a write operation, it is easy to see that ι and the re-fetch operation can be swapped. If the label ι is a write operation, it must not interfere the re-fetch operation. Because any re-fetch operations can only read versions written before the phase change step. The full proof of the right mover is given in Theorem A.13 on page 205.

Assign the new trace to ζ_1 and go back to step (1). There are finite number of transitions in a trace. After each iteration of step (1), a re-fetch transition of a transaction moves closer to the end label of the transaction. Therefore, step (1) must terminate. The final trace ζ_1 satisfies Eq. (6.5).

(2) **Left mover: out-of-order writes**. Given a trace ζ_1 satisfying Eq. (6.5), take two out-of-order write operations with the shortest distance in between, $(cl, r, (\mathtt{W}, k, v), id, d)$ and $(cl', r', (\mathtt{W}, k', v'), id', d')$, and the operation immediately before the latter write, ι:

$$\zeta_1 = \zeta' \xrightarrow{(cl,r,(\mathtt{W},k,v),id,d)} \zeta'' \xrightarrow{(cl',r',(\mathtt{W},k',v'),id',d')} \zeta''' \wedge id' \sqsubseteq id,$$

Note that there is no other out-of-order in the trace segment ζ''. We show that any step ι in ζ'' that may interfere $(cl', r', (\mathtt{W}, k', v'), id', d')$ can be moved before $(cl, r, (\mathtt{W}, k, v), id, d)$. Then we move the out-of-order write $(cl', r', (\mathtt{W}, k', v'), id', d')$ before $(cl, r, (\mathtt{W}, k, v), id, d)$.

We immediately have $cl \neq cl'$ and $r \neq r'$, because the replica time must monotonically increase (detail is given in Prop. A.14 on page 206). Let first consider any step with label ι in the trace segment ζ''. The step ι cannot be a write step. Assume $\iota = (cl'', r'', (\mathtt{W}, k'', v''), id'', d'')$. Since all version identifier is unique and ordered (detail is given in Prop. A.14 on page 206), this means that (i) $id'' \sqsubseteq id' \sqsubseteq id$; (ii) $id' \sqsubseteq id'' \sqsubseteq id$; or (iii) $id' \sqsubseteq id \sqsubseteq id''$. However, all the cases above contradict that we pick two out-of-order write with the shortest distance. Hence, ι can be snapshot, optimistic read, phase change, re-fetch read and commit steps for multi-reads, and synchronisation.

Second, we focus on the first step ι that are from cl' and r' in the trace segment ζ''. The label ι cannot depends on $(cl, r, (\mathtt{W}, k, v), id, d)$. If ι is a synchronisation step, it must be the case that $\iota = (r', id'')$ such that $id'' \sqsubseteq id'$, because the replica time must monotonically increase (detail is given in Prop. A.14 on page 206). This means $id'' \sqsubseteq id' \sqsubseteq id$. Let consider ι be a step for multi-reads from r' and a client cl'' (cl'' might be cl'). If ι is a local computation including snapshot, phase change and commit steps, it is trivial ι does not depend on $(cl, r, (\mathtt{W}, k, v), id, d)$. If ι is a optimistic read or re-fetch read, that reads from a version of a key k'' identified by id''. The step ι does not depend on $(cl, r, (\mathtt{W}, k, v), id, d)$ in the sense that $\neg\left((k, id) \xrightarrow{\mathsf{DEP}} (k'', id'')\right)$. Assume $(k, id) \xrightarrow{\mathsf{DEP}} (k'', id'')$. It means that $id \sqsubseteq id''$. Because the replica time must monotonically increase (detail is given in

Prop. A.14 on page 206), then $id'' \sqsubseteq id'$. Hence, $id \sqsubseteq id'$, which leads to contradiction. Given above, it allows us to inductively move the first ι of a step for multi-reads from r' and a client cl'' in the trace segment ζ'' before the step $(cl, r, (\mathtt{W}, k, v), id, d)$. This yields a new trace:

$$\left(\zeta' \Rightarrow \zeta_3 \xrightarrow{(cl,r,(\mathtt{W},k,v),id,d)} \zeta_2 \xrightarrow{\iota_1} \_ \xrightarrow{(cl',r',(\mathtt{W},k',v'),id',d')} \zeta'''\right) \simeq \zeta_1 \wedge id' \sqsubseteq id$$

where the prefix of ζ_3 is ζ' and followed by steps that are from cl' and r', and ζ_2 contains the rest steps in ζ'. Note that ζ_2 contains any steps in ζ'' that are not from cl' nor r',

Now consider the last step ι' of ζ_2, which must from a client cl'' and a replica r'' that are different from cl' and r' respectively. because different replicas and clients do not interfere with each others, it is easy to see that ι' can be move to right, delaying this step. In other words, the out-of-order write operation can be moved to the left. We perform this until the two out of order writes are adjacent:

$$\left(\zeta' \Rightarrow \zeta_3 \xrightarrow{(cl,r,(\mathtt{W},k,v),id,d)} \_ \xrightarrow{(cl',r',(\mathtt{W},k',v'),id',d')} \zeta_4\right) \simeq \zeta_1 \wedge id' \sqsubseteq id$$

where ζ_4 contains steps in ζ_2 and then steps in ζ'''. We now can swap the two out-of-order writes:

$$\left(\zeta' \Rightarrow \zeta_3 \xrightarrow{(cl',r',(\mathtt{W},k',v'),id',d')} \_ \xrightarrow{(cl,r,(\mathtt{W},k,v),id,d)} \zeta_4\right) \simeq \zeta_1 \wedge id' \sqsubseteq id.$$

Assign the new trace to ζ_1 and go back to step (2). For each iteration of step (2), two out-of-order write operations move closer to each other and eventually swap position. Because there are finite number of transitions in a trace and thus finite number of out-of-order write operations, then step (2) must terminate. The full proof of this left mover is given in Theorem A.15 on page 206. The final trace ζ_1 satisfies Eq. (6.4).

After steps (1) and (2), the trace ζ_1 is a normalised COPS trace as $\mathtt{NCOPSTrace}\,(\zeta_1)$. ■

In our kv-store semantics (Chapter 4), we assign a unique identifier for every transaction in CAtomicTrans in Fig. 4.2. In COPS semantics, there is no explicit transaction identifier. We encode the COPS version identifier as *COPS transaction identifier* for the single-write transaction who committed the version. We then annotate multiple-read transactions with transaction identifiers that preserves the session order using the read-only transaction counter m in $\hat{t}_{cl}^{(n,r,m)}$.

Definition 6.13 (COPS transaction identifiers)**.** *The set of* COPS transaction identifiers, *COPSTxID* $\ni \hat{t}$, *is defined by:*

$$\mathit{COPSTxID} \stackrel{def}{=} \left\{ \hat{t}_{cl}^{(n,r,m)} \;\middle|\; cl \in \mathit{CID} \wedge r \in \mathit{COPSRep} \wedge n, m \in \mathbb{N} \right\} \cup \{t_0\}\,.$$

The order over transactions is defined by:

$$\hat{t}_{cl}^{(n,r,m)} \sqsubseteq \hat{t}_{cl'}^{(n',r',m')} \overset{def}{\Leftrightarrow} n < n' \vee (n = n' \wedge (r < r' \vee (r = r' \wedge m < m'))).$$

The COPS transaction identifier $\hat{t}_{cl}^{(n,r,m)}$ contains a client identifier cl, a local time n, a replica identifier r and an extra counter m for read-only transactions. For the single write transaction, the transaction identifier is the version identifier (n, r) and $m = 0$. we then use the extra counter m to annotate read-only transactions. Note that, transaction identifiers are order lexicographically. If we encode all tuples, (r, n, m), as individual numbers, all these encodes numbers must be totally ordered. Therefore, there is a bijection between COPSTxID and kv-store transaction identifiers TxID defined in Def. 4.1.

We first assign COPS transaction identifiers to single-write transactions: that is, we extend the version identifiers with the client identifiers and the reader counters m being zero. We then assign COPS transaction identifiers with non-zero read-only counters to multiple-read transactions: that is, for each multiple-read transaction, we annotate the re-fetch operations and the end operation with the next available identifier in $\hat{\mathcal{E}} \in \text{CID} \to \text{COPSTxID}$. To preserve the session order, we update $\hat{\mathcal{E}}$ for the client cl, if cl commits a new transaction. By construction, the transaction identifiers that are assigned to multiple-read transactions must be unique. The detail is given in Prop. A.17 on page 208.

Definition 6.14 (Annotated normalised COPS traces)**.** *A* client local time environment *is defined by:* $\hat{\mathcal{E}} \in \mathit{CID} \to \mathit{COPSTxID}$*. Given a normalised COPS trace* $\zeta \in \mathit{ANCOPSTrace}$*, the annotated normalised COPS trace is defined by:*

$$\mathsf{COPSToExt}\left((\Theta_0, \hat{\mathtt{P}}_0), \hat{\mathcal{E}}\right) \overset{def}{=} (\Theta_0, \hat{\mathtt{P}}_0),$$

$$\mathsf{COPSToExt}\left(\left(\left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I} \xrightarrow{\iota} \zeta\right), \hat{\mathcal{E}}\right)$$

$$\overset{def}{=} \begin{cases} \left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I} \xrightarrow{\iota, \hat{t}_{cl}^{(n,r,0)}} \mathsf{COPSToExt}\left(\zeta, \hat{\mathcal{E}}\left[cl \mapsto \hat{t}_{cl}^{(n,r,1)}\right]\right) & \text{if } \iota = (cl, r, (\mathtt{W}, k, v), (n, 0), d), \\ \left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I} \xrightarrow{\iota, \hat{\mathcal{E}}(cl)} \mathsf{COPSToExt}\left(\zeta, \hat{\mathcal{E}}\right) & \text{if } \iota = \left(cl, r, (\mathtt{R}, k, v), \hat{t}, d, \mathtt{Ref}\right), \\ \left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I} \xrightarrow{\iota, \hat{\mathcal{E}}(cl)} \mathsf{COPSToExt}\left(\zeta, \hat{\mathcal{E}}\left[cl \mapsto \hat{t}_{cl}^{(n,r,m+1)}\right]\right) & \text{if } \iota = (cl, r, \hat{u}, \mathtt{E}) \wedge \hat{\mathcal{E}}(cl) = \hat{t}_{cl}^{(n,r,m)}, \\ \left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I} \xrightarrow{\iota} \mathsf{COPSToExt}\left(\zeta, \hat{\mathcal{E}}\right) & \text{otherwise.} \end{cases}$$

For each single-write transaction, we explicitly annotate it with a transaction identifier, containing the new version identifier and the client identifier. Moreover, for each read-only transaction, we explicitly assign it with a transaction identifier, and use the reader counter to encoding the session order SO. Note that the COPS transaction identifiers of the forms $\hat{t}_{cl}^{(n,r,m)}$ is isomorphic to the transaction identifiers in kv-store (Def. 4.1).

6.1.4 Verification: Trace Refinement

We show how to encode a COPS machine state in our centralised kv-store, and client contexts as views on the kv-store. Given a kv-store program trace η encoding a COPS trace, we then show that η can be obtained under the execution test of causal consistency.

For each annotated normalised COPS trace, the final configuration corresponds to a kv-store by replaying all the transactions on the initial kv-store. This means:

(1) For a single-write transaction, it appends a new version to the $\mathcal{K}$ with the writer, being the annotated transaction identifier defined in Def. 6.14 and the reader set, being the empty set (the first case in the induction case of $\mathsf{COPSToKVS}$ in Fig. 6.5); and

(2) For a read-only transaction, it adds the annotated transaction identifier to the reader sets of versions being read (the second case in the induction case of $\mathsf{COPSToKVS}$ in Fig. 6.5).

Note that, the final configuration itself contains enough information for all the write transactions, however, we need the trace to recover the annotated identifiers for multiple-read transactions.

Definition 6.15 (Centralised COPS kv-store)**.** *Given an annotated normalised COPS trace $\zeta \in$ ANCOPSTrace the centralised kv-store induced by the trace, written $\mathsf{COPSToKVS}\,(\zeta)$, is defined in Fig. 6.5.*

Recall the properties of annotated normalised COPS traces:

(1) multiple-read transactions are annotated with unique identifiers (detail in Prop. A.17 on page 208); and

(2) it is safe to append new versions (detail in Prop. A.18 on page 209).

This mean that the kv-store $\mathcal{K} = \mathsf{COPSToKVS}\,(\zeta)$ is well-formed (defined in Def. 4.5). Also by the notion of $\mathsf{COPSToKVS}$, the kv-store only contains versions in ζ and vice versa. The detail is given in Prop. A.19 on page 209.

Given the kv-store, $\mathcal{K}$, encoding the final configuration of an annotated normalised COPS trace, the final context $\hat{u}$ for every client can be encoded to a view u on $\mathcal{K}$ in the sense that the view u contains all versions ν that are included in $\hat{u}$ and versions that ν depends on, directly or indirectly.

Definition 6.16 (COPS context views)**.** *Assume an annotated normalised COPS trace ζ. Given the last configuration $((\mathcal{R},\hat{\mathcal{U}}),\_) = \mathsf{Last}\,(\zeta)$ and the kv-store $\mathcal{K} = \mathsf{COPSToKVS}\,(\zeta)$, the*

$$\mathsf{COPSToKVS}\left(\Theta_0, \hat{\mathtt{P}}\right) \stackrel{def}{=} \mathcal{K}_0,$$

$$\mathsf{COPSToKVS}\left(\zeta \xrightarrow{\iota} \left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I}\right) \stackrel{def}{=} \mathtt{let}\ \mathcal{K} = \mathsf{COPSToKVS}\left(\zeta\right)\ \mathtt{in}$$

$$\begin{cases} \mathcal{K}\left[k \mapsto \mathcal{K}\left(k\right) :: \left[\left(v, \hat{t}^{(n,r,0)}_{cl}, \emptyset\right)\right]\right] & \text{if } \iota = \left(cl, r, \left(\mathtt{W}, k, v\right), \left(n, r\right), d\right), \hat{t}^{(n,r,0)}_{cl} \\ \mathcal{K}\left[k \mapsto \mathcal{K}\left(k\right)\left[i \mapsto \left(v, \hat{t}^{(n,r',0)}_{cl'}, T \uplus \hat{t}'\right)\right]\right] & \text{if } \iota = \left(\left(cl, r, \left(\mathtt{R}, k, v\right), \left(n, r'\right), d, \mathtt{Ref}\right), \hat{t}'\right), \wedge \mathcal{K}\left(k\right)_{|i} = \left(v, \hat{t}^{(n,r',0)}_{cl'}, T\right), \\ \mathcal{K} & \text{otherwise.} \end{cases}$$

$$\mathsf{COPSToKVTrace}\left(\Theta_0, \hat{\mathtt{P}}\right) \stackrel{def}{=} \mathtt{let}\ s_0 = \lambda \mathtt{x} \in \mathrm{V}_{\mathrm{AR}}.\ 0\ \mathtt{in}\left(\mathsf{COPSToKVS}\left(\Theta_0, \hat{\mathtt{P}}\right), \mathsf{COPSViews}\left(\Theta_0, \hat{\mathtt{P}}\right), \lambda cl \in \mathsf{Dom}\left(\hat{\mathtt{P}}\right).\ s_0\right), \mathsf{COPSToKVProg}\left(\hat{\mathtt{P}}\right)$$

$$\mathsf{COPSToKVTrace}\left(\zeta \xrightarrow{\iota} \left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I}\right) \stackrel{def}{=}$$

$$\mathtt{let}\ \left(\left(\mathcal{K}, \mathcal{U}, \mathcal{E}\right), \mathtt{P}\right) = \mathsf{Last}\left(\mathsf{COPSToKVTrace}\left(\zeta\right)\right),\ \mathcal{K}' = \mathsf{COPSToKVS}\left(\zeta \xrightarrow{\iota} \left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I}\right),$$

$$\mathcal{U}' = \mathsf{COPSViews}\left(\mathcal{R}, \hat{\mathcal{U}}\right),\ \text{and}\ \mathtt{P}' = \mathsf{COPSToKVProg}\left(\mathtt{I}\right)\ \mathtt{In}$$

$$\begin{cases} \mathsf{COPSToKVTrace}\left(\zeta\right) \xrightarrow{(cl, \mathcal{U}(cl), \mathcal{F})}_{\top} \left(\mathcal{K}', \mathcal{U}', \mathcal{E}\right), \mathtt{P}' & \text{if } \iota = \left(cl, r, \left(\mathtt{W}, k, v\right), id, d\right), \hat{t}, \wedge \mathcal{F} = \left\{\left(\mathtt{W}, k, v\right)\right\} \\ \mathsf{COPSToKVTrace}\left(\zeta\right) & \text{if } \iota = \left(cl, r, \mathtt{S}\right) \vee \iota = \left(cl, r, \left(\mathtt{R}, k, v\right), id, d, \mathtt{Opt}\right) \vee \iota = \left(r, id\right) \end{cases}$$

$$\mathsf{COPSToKVTrace}\left(\zeta \xrightarrow{\iota} \zeta' \xrightarrow{\iota'} \left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I}\right) \stackrel{def}{=}$$

$$\mathtt{let}\ \left(\left(\mathcal{K}, \mathcal{U}, \mathcal{E}\right), \mathtt{P}\right) = \mathsf{Last}\left(\mathsf{COPSToKVTrace}\left(\zeta\right)\right),\ \mathcal{K}' = \mathsf{COPSToKVS}\left(\zeta \xrightarrow{\iota} \zeta' \xrightarrow{\iota'} \left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I}\right),$$

$$\mathcal{U}' = \mathsf{COPSViews}\left(\mathcal{R}, \hat{\mathcal{U}}\right),\ \text{and}\ \mathtt{P}' = \mathsf{COPSToKVProg}\left(\mathtt{I}\right)\ \mathtt{in}\ \mathsf{COPSToKVTrace}\left(\zeta\right) \xrightarrow{(cl, \mathcal{U}'(cl), \mathcal{F})}_{\top} \left(\mathcal{K}', \mathcal{U}', \mathcal{E}'\right), \mathtt{P}'$$

$$\mathtt{where}\ \iota = \left(cl, r, \mathtt{P}\right) \wedge \zeta' = \xrightarrow{(cl, r, (\mathtt{R}, k_0, v_0), id_0, d_0, \mathtt{Ref}), \hat{t}}_{\cdots} \xrightarrow{(cl, r, (\mathtt{R}, k_n, v_n), id_n, d_n, \mathtt{Ref}), \hat{t}}_{-}$$
$$\wedge\ \iota' = \left(cl, r, \hat{u}', \mathtt{E}\right), \hat{t} \wedge \mathcal{F} = \left\{\left(\mathtt{R}, k_0, v_0\right), \cdots, \left(\mathtt{R}, k_n, v_n\right)\right\} \wedge \mathcal{E}' = \mathcal{E}\left[\mathtt{x}_0 \mapsto v_0\right] \cdots \left[\mathtt{x}_n \mapsto v_n\right]$$

Figure 6.5: Definitions of COPSToKVS and COPSToKVTrace

view environment induced by the client context $\hat{\mathcal{U}}$*, written* $\mathsf{COPSViews}\left(\mathcal{R},\hat{\mathcal{U}}\right)$*, is defined by:*

$$\mathsf{COPSViews}\left(\mathcal{R},\hat{\mathcal{U}}\right)(cl) \stackrel{def}{=} \lambda k \in \mathit{KEY}.\left\{ i \;\middle|\; \begin{array}{l} \exists k' \in \mathit{KEY}.\, \exists id, id' \in \mathit{COPSVERID}. \\ (k', id') \in \hat{\mathcal{U}}(cl) \wedge (k, id) \xrightarrow{\mathsf{DEP}^*_{\mathcal{R}}} (k', id') \\ \qquad\qquad \wedge\, id' = \mathsf{IdOf}(\mathcal{K}(k, i)) \end{array} \right\} \cup \{0\},$$

where the COPS dependency relation*,* $\mathsf{DEP}_{\mathcal{R}}$*, is defined by:*

$$\mathsf{DEP}_{\mathcal{R}} \stackrel{def}{=} \bigcup_{(\mathcal{K},\_)\in\mathsf{Image}(\mathcal{R})} \left\{ ((k, id), (k', id')) \;\middle|\; \begin{array}{l} \exists i.\, id' = \mathsf{IdOf}(\mathcal{K}(k', i)) \\ \quad \wedge\, (k, id) \in \mathsf{DepSetOf}(\mathcal{K}(k', i)) \end{array} \right\}.$$

The relation $\mathsf{DEP}_{\mathcal{R}}$ on a COPS database $\mathcal{R}$ denotes dependency relations between versions. If $((k, id), (k', id')) \in \mathsf{DEP}_{\mathcal{R}}$, then (k, id) is included in the dependent set of the version identified by id'. Since all versions have unique transactions, it is easy to see that the view induced by the COPS client context is well-formed. The detail is given in Prop. A.20 on page 210.

Last, it is easy to convert the COPS syntax, namely `put` and `read` APIs, to transactional syntax in our kv-store semantics.

Definition 6.17 (COPS atomic transactions)**.** *Given a COPS command* $\hat{\mathtt{C}} \in \mathit{COPSCMD}$*, the kv-store command,* $\mathsf{COPSToKVCmd} : \mathit{COPSRunCommands} \to \textsc{Commands}$*, induced by a COPS runtime command is defined by:*

$$\begin{aligned}
\mathsf{COPSToKVCmd}(\mathtt{put}(k, v)) &\stackrel{def}{=} [\ \mathtt{wt}(k, v)\ ], \\
\mathsf{COPSToKVCmd}(\mathtt{read}([k_0, \cdots, k_n])) &\stackrel{def}{=} [\ \mathtt{x}_0 := \mathtt{rd}(k_0)\,; \cdots; \mathtt{x}_n := \mathtt{rd}(k_n)\ ], \\
\mathsf{COPSToKVCmd}\left(\mathtt{read}([k_0, \cdots, k_n]) : \hat{\mathcal{V}}\right) &\stackrel{def}{=} [\ \mathtt{x}_0 := \mathtt{rd}(k_0)\,; \cdots; \mathtt{x}_n := \mathtt{rd}(k_n)\ ], \\
\mathsf{COPSToKVCmd}\left(\mathtt{read}([k_0, \cdots, k_n]) : (\hat{\mathcal{V}}, D)\right) &\stackrel{def}{=} [\ \mathtt{x}_0 := \mathtt{rd}(k_0)\,; \cdots; \mathtt{x}_n := \mathtt{rd}(k_n)\ ], \\
\mathsf{COPSToKVCmd}\left(\mathtt{read}([k_0, \cdots, k_n]) : (\hat{\mathcal{V}}, D, \hat{\mathcal{V}}')\right) &\stackrel{def}{=} [\ \mathtt{x}_0 := \mathtt{rd}(k_0)\,; \cdots; \mathtt{x}_n := \mathtt{rd}(k_n)\ ], \\
\mathsf{COPSToKVCmd}(\mathtt{R}\,;\mathtt{R}') &\stackrel{def}{=} \mathsf{COPSToKVCmd}(\mathtt{R})\,;\mathsf{COPSToKVCmd}(\mathtt{R}').
\end{aligned}$$

The kv-store program induced by a COPS runtime program is defined by:

$$\mathsf{COPSToKVProg}(\mathtt{I}) \stackrel{def}{=} \lambda cl \in \mathsf{Dom}(\mathtt{I}).\, \mathsf{COPSToKVCmd}\left(\hat{\mathtt{P}}(cl)\right).$$

We now encode the COPS trace ζ into a kv-store program trace η. Recall that we encode runtime information in the syntax of the read-only transaction API; this information is transparent to clients in our kv-store operational semantics.

Definition 6.18 (COPS kv-store traces)**.** *Given an annotated normalised COPS trace* $\zeta \in \mathit{ANCOPSTRACE}$*, the kv-store traces induced by* ζ*, written* $\mathsf{COPSToKVTrace}(\zeta)$*, is defined in*

Fig. 6.5.

We convert a COPS trace ζ to a kv-store trace η, in the definition of $\mathsf{COPSToKVTrace}$ in Fig. 6.5.

(1) The base case of $\mathsf{COPSToKVTrace}$ convert the initial COPS configuration to an initial kv-store configuration: both configurations contain the same amount of clients, and stacks in kv-store semantics being initialised to all zero.

(2) In the first inductive case of $\mathsf{COPSToKVTrace}$, we convert all steps except re-fetch read steps. Assume a COPS trace

$$\zeta \xrightarrow{\iota} \left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I}$$

and $((\mathcal{K}, \mathcal{U}, \mathcal{E}), \mathtt{P}) = \mathsf{Last}\left(\mathsf{COPSToKVTrace}\left(\zeta\right)\right)$ being the last configuration in ζ. Let $\eta = \mathsf{COPSToKVTrace}\left(\zeta\right)$ be the kv-store trace that encodes ζ. If the next step ι is start of read-only transaction $\mathtt{S}$, optimists read $\mathtt{Opt}$ and synchronisation step, the new kv-store trace is exactly η, because these steps are transparent to clients. If the new step is a single-write transaction $(cl, r, (\mathtt{W}, k, v), id, d)$, then we simulate this transaction in the kv-store trace, that is

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

(3) In the second inductive case of $\mathsf{COPSToKVTrace}$, we convert the read-only transaction. Assume a COPS trace

$$\zeta \xrightarrow{\iota} \zeta' \xrightarrow{\iota'} \left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I}$$

and $((\mathcal{K}, \mathcal{U}, \mathcal{E}), \mathtt{P}) = \mathsf{Last}\left(\mathsf{COPSToKVTrace}\left(\zeta\right)\right)$ being the last configuration in ζ. Assume that cl is the next scheduled client. Recall that we ignore all optimistic read operations in the first phase, and by the definition of normalised traces in Def. 6.11, all re-fetch reads from a transaction $\hat{t}$ must be wrapped between the phase change step ι labelled with $\mathtt{P}$ and end step ι' labelled with $\mathtt{E}$. In the definition, we collect all the re-fetch read operations in ζ' as the fingerprint $\mathcal{F}$ of the transaction $\hat{t}$ in kv-store trace. We use the view $u = \mathcal{U}'(cl)$ that encodes the *new* client context after the read (noting $\mathcal{U}' = \mathsf{COPSViews}\left(\mathcal{R}, \hat{\mathcal{U}}\right)$), as the view for $\hat{t}$, which guarantees the well-formed of the trace: the transaction reads the latest versions in the view u. The view after the update simply remains the same as u. Therefore we have the new kv-store trace

$$\mathsf{COPSToKVTrace}\left(\zeta\right) \xrightarrow{(cl, \mathcal{U}'(cl), \mathcal{F})}_{\top} (\mathcal{K}', \mathcal{U}', \mathcal{E}'), \mathtt{P}'$$

Given kv-store program traces η under the read atomic $\top$ that encode COPS traces, we prove that every step in these traces satisfies the execution test for $\mathtt{CC}$.

Theorem 6.19 (COPS causal consistency)**.** *Given an annotated normalised COPS trace $\zeta \in$ ANCOPSTrace, the kv-store traces, $\eta = \mathsf{COPSToKVTrace}\left(\zeta\right)$, can be obtained under $\mathtt{ET}_{\mathtt{CC}}$.*

Proof sketch. By the definition of $\mathtt{ET}_{\mathtt{CC}}$, we prove $\mathtt{ET}_{\mathtt{MR}}$, $\mathtt{ET}_{\mathtt{RYW}}$ and $\mathtt{PreClosed}\,(\mathcal{K}, u, \mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO})$ separately. It is easy to prove every step satisfies $\mathtt{ET}_{\mathtt{MR}}$ and $\mathtt{ET}_{\mathtt{RYW}}$, because a client context monotonic increases and always contains versions written by the client itself. In Prop. 6.20, we show that the relation $\mathsf{DEP}^*_{\mathcal{R}}$ contains all edges in the relation $(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO})^+$; this implies that the any view u that encodes a client context, must be always close with respect to $(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO})^+$, and hence predicate $\mathtt{PreClosed}\,(\mathcal{K}, u, \mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO})$ holds. The detail is given in appendix A.6 on page 210. ■

Proposition 6.20 (COPS dependency relation to CC relation)**.** *Given an annotated normalised COPS trace $\zeta \in$ ANCOPSTrace and the kv-store program trace $\eta = \mathsf{COPSToKVTrace}\,(\zeta)$, let $\mathcal{K}$ be the final kv-store such that $(\mathcal{K}, \_, \_) = \mathsf{Last}\,(\eta)$ and $\mathcal{R}$ be the final state of COPS database such that $(\mathcal{R}, \_, \_) = \mathsf{Last}\,(\zeta)$. Given two versions $\mathcal{K}\,(k, i)$ written $\hat{t}$ and $\mathcal{K}\,(k', i')$ written by $\hat{t}'$, if $\hat{t}^{(n,r,0)}_{cl} \xrightarrow{(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO})^+} \hat{t}^{(n',r',0)}_{cl'}$ then $(k, (n, r)) \xrightarrow{\mathsf{DEP}^*_{\mathcal{R}}} (k', (n', r'))$.*

Proof. Let $\hat{t} = \hat{t}^{(n,r,0)}_{cl}$ and $\hat{t}' = \hat{t}^{(n',r',0)}_{cl'}$. By the hypothesis, both transactions are single-write transactions, which means that $(\hat{t}, \hat{t}') \notin \mathsf{WR}_{\mathcal{K}}$. For any transaction in COPS, it either is a read-only transaction or a single-write transaction. Recall that SO is transitive. This means that $\hat{t} \xrightarrow{(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO})^+} \hat{t}'$ if and only if $\hat{t} \xrightarrow{(\mathsf{WR}^?_{\mathcal{K}}; \mathsf{SO})^+} \hat{t}'$ and thus it is sufficient to prove the following result

$$\hat{t} \xrightarrow{\mathsf{WR}^?_{\mathcal{K}}; \mathsf{SO}} \hat{t}' \Rightarrow (k, (n, r)) \xrightarrow{\mathsf{DEP}^*_{\mathcal{R}}} (k', (n', r')). \tag{6.6}$$

(1) [**Case**: $\hat{t} \xrightarrow{\mathsf{SO}} \hat{t}'$] Assume that $\hat{t}, \hat{t}'$ are from client cl that interacts with replica r. For this case, transaction $\hat{t}$ must commit before $\hat{t}'$: that is

$$\zeta = \cdots \xrightarrow{(cl, r, (\mathtt{W}, k, v), (n, r), d), \hat{t}} \cdots \xrightarrow{(cl, r, (\mathtt{W}, k', v'), (n', r'), d'), \hat{t}'} \cdots.$$

By rule COPSWrite and the COPS invariant that replica time monotonically increases (detail is given in Prop. A.14 on page 206), it follows that $(k, (r, n)) \in d'$ which implies Eq. (6.6).

(2) [**Case**: $\hat{t} \xrightarrow{\mathsf{WR}_{\mathcal{K}}} \hat{t}'' \xrightarrow{\mathsf{SO}} \hat{t}'$ **for a read-only transaction** $\hat{t}''$] Assume that $\hat{t}'', \hat{t}'$ are from client cl that interacts with replica r. For this case, we know

$$\zeta = \cdots \xrightarrow{(cl, r, (\mathtt{R}, k, v), (n, r), d, \mathtt{Ref}), \hat{t}''} \cdots \xrightarrow{(cl, r, \hat{u}, \mathtt{E}), \hat{t}''} \cdots \xrightarrow{(cl, r, (\mathtt{W}, k', v'), (n', r'), d'), \hat{t}'} \cdots.$$

By rules COPSRefetch and COPSFinish, $(k, (n, r)) \in d$ and then by the COPS invariant that replica time monotonically increases (detail is given in Prop. A.14 on page 206), it follows that $(k, (n, r)) \in d'$ which implies Eq. (6.6). ■

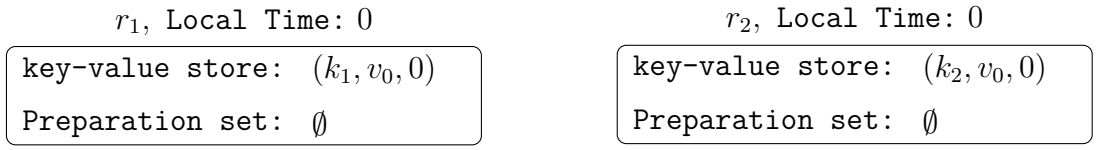


Figure 6.6: An initial Clock-SI state with two shards, r_1 and r_2, which contains keys k_1 and k_2 respectively

6.2 Verification of Clock-SI Protocol

Clock-SI [Du et al., 2013] is a partitioned distributed NoSQL database, in which each site, also called *shard*, contains a distinct partition of keys. Clock-SI implements snapshot isolation: (1) a simplified version where each session contains only one transaction and (2) a full version where each session can have multiple transactions. Here we verify the full version. The key idea of Clock-SI is to extend the timestamp-based, centralised implementation for `SI` initially presented in [Berenson et al., 1995]. In Clock-SI, all servers, or shards, do not agree on a global physical time, however, the time diffidence between shards is often bound. Therefore, a transaction t in Clock-SI takes the local time of a shard as the transaction start time. This transaction t moves between shards to read keys, however, t can only progress when the shards local time are later than the transaction start time, otherwise t waits. At the end of t, it commits all the write operations to the appropriate shards, using a two-phase commit protocol. Under this protocol, a transaction is allowed to commit only if there is another transaction that concurrently writes to the same key.

We give an informal description of Clock-SI protocol in Section 6.2.1 and a formal, fine-grained operational semantics in crefsec:clock-si-model,sec:clock-si-semantics. Similar to the verification of COPS, we introduce the normalised Clock-SI traces in Section 6.2.4, where transactions in the traces are not interleaved. We then convert normalised Clock-SI traces to kv-store traces and show the kv-store traces satisfies the execution test for `SI`.

Notation. We pick the same letters for the concepts that have similar meaning with respect to those in our kv-store operational semantics, and annotate them with *tilde*, for example, Clock-SI key-value store $\tilde{\mathcal{K}}$.

6.2.1 Clock-SI protocol

Clock-SI is a fully partitioned distributed key-value database, with each shard storing all history version of a distinct part of keys as shown in Figs. 6.6 to 6.9. Each version in a shard, for example $(k_1, v_0, 0)$ in Fig. 6.6, consists of a key, k_1, a value, v_0, and the time this version has been committed, $n = 0$. A transaction in Clock-SI contains arbitrary read and write operations,

and internal computation. For example, the following program $\mathtt{P}_{\mathtt{Clock}}$, contains two clients, each of which has one transaction. Both the transactions read the value of keys k_1 and k_2, and if the values are the initial value v_0, they updates k_1 or k_2 respectively.

$$\mathtt{P}_{\mathtt{Clock}} \stackrel{def}{=} cl_1 : \begin{bmatrix} \mathtt{x} := \mathtt{rd}\,(k_1)\,;\mathtt{y} := \mathtt{rd}\,(k_2)\,; \\ \mathtt{if}\,(\mathtt{x} = v_0\ \&\&\ \mathtt{y} = v_0)\ \ \mathtt{wt}\,(k_2, v') \end{bmatrix} \Big\| cl_2 : \begin{bmatrix} \mathtt{z} := \mathtt{rd}\,(k_1)\,;\mathtt{m} := \mathtt{rd}\,(k_2)\,; \\ \mathtt{if}\,(\mathtt{z} = v_0\ \&\&\ \mathtt{m} = v_0)\ \ \mathtt{wt}\,(k_1, v) \end{bmatrix}.$$

The idea of Clock-SI is that:

(1) a transaction t tracks a *transaction snapshot time* $\mathsf{STimeOf}\,(t) = n$, and when reading a key k, the transaction reads the latest value for k before n; and

(2) when committing, the transaction uses a two-phase commit protocol in that new versions first commit to the *preparation sets* of the shards, as shown in Fig. 6.8, and then commit to key-value stores.

Each shard r tracks a local time, which is the actual physical time of the shard. In an ideal situation, local times in different shards should be the same. However, this is impossible in practice. A client also carries a local time that tracks the commit time of its last transaction, which is used to maintain the session order over transactions from the client.

Let us describe the protocol using the program $\mathtt{P}_{\mathtt{Clock}}$. A client commits a transaction and the client local time as the minimum transaction snapshot time to an *arbitrary* shard, and awaiting confirmation. Upon receiving the transaction t, the shard r acts as the coordinator of the transaction:

(1) r decides the transaction snapshot time of t;

(2) r executes the commands of t and tracks the effect of these commands in a *read-write set*; and

(3) r commits the effect of t, all new values, to the appropriate shards using a two-phase commits protocol.

Assume that cl_1 commits its transaction t_1 to shard r_1 with client local time 0. This local time is the transaction snapshot time for the new transaction. Upon receiving the new transaction, coordinator r_1 assigns the coordinator local time n as $\mathsf{STimeOf}\,(t_1)$, only if n is greater than the minimum transaction snapshot time. For example, in Fig. 6.7, r_1 waits until the local time becomes 1, and then assigns 1 as $\mathsf{STimeOf}\,(t_1)$. Similarly, r_2 assigns 1 as $\mathsf{STimeOf}\,(t_2)$ in parallel. The coordinator also initialises the read-write set to be the empty set. The read-write set is the same as the fingerprint in our kv-store semantics, which tracks the observable effect of the transaction, containing at most one read and one write operation of each key.

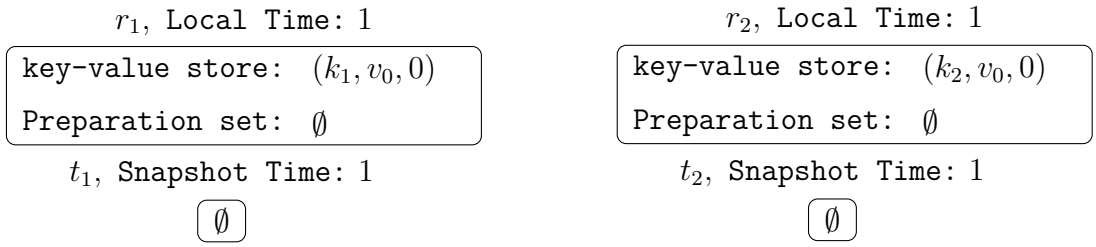


Figure 6.7: Shard r_1 assigns the snapshot time to transaction t_1 and shard r_2 assigns the snapshot time to transaction t_2 in parallel

After the initialisation, it is ready to execute the transaction. Let us take t_1 as an example. For any lookup command, for example $\mathtt{x} := \mathtt{rd}\,(k_1)$, the coordinator first checks if there is an entry of k_1 in the read-write set of t_1. If there is a write operation of k_1, for example $(\mathtt{W}, k_1, v)$, it means that the transaction has updated the key internally. If there is a read operation of k_1, for example $(\mathtt{R}, k_1, v)$, it means that the transaction has read the key before. In both cases, because snapshot isolation satisfies snapshot property, the transaction reads value v in the previous write operation or in the previous read operation from the read-write set respectively. If there is no entry of k_1, the coordinator sends a request to the shard r_1 containing key k_1, and $\mathsf{STimeOf}\,(t_1)$, which is used to fetch to correct value. Upon receiving the request, shard r_1 waits until:

(1) the shard-local time is greater than $\mathsf{STimeOf}\,(t_1)$; and

(2) there is no version of k_1 in the preparation set that has time-stamp smaller than $\mathsf{STimeOf}\,(t_1)$.

If both conditions are satisfied, then there is no new version of k_1 that might commit before $\mathsf{STimeOf}\,(t_1)$. Shard r_1 then replies with the latest value of k_1 before $\mathsf{STimeOf}\,(t_1)$. For any mutation command, for example $k_2 := \mathtt{rd}\,(v')$, the coordinator erases any previous write operations of k_2 and adds a new write operation $(\mathtt{W}, k_2, v')$. Note that different transactions might execute in parallel as shown in Fig. 6.8a.

When a transaction reaches the end, `skip`, the final read-write set contains the effect of this transaction, for example the read-write set for t depicted in Fig. 6.8a. The coordinator then commits any updates in the read-write set to appropriate shards, using a two-phase commits protocol. In the first commit phase, for any write operation such as $(\mathtt{W}, k_2, v')$ in Fig. 6.8b, the coordinator sends a commit request to the shard that contains the key. Upon receiving the commit request, the shard will accept the new version if there is no conflicting write: that is, any versions of the key that has been committed or prepared in the shard since $\mathsf{STimeOf}\,(t_1)$. For example in Fig. 6.8a, the new value v' for k_2 can commit in shard r_2, since there are no other versions of k_2 with time-stamp greater than $\mathsf{STimeOf}\,(t_1) = 1$. If there is no conflicting write, the shard assigns the shard-local time as the preparation time of the new version, and puts the version in the preparation set as shown in Fig. 6.8b. In the successful case, the shard

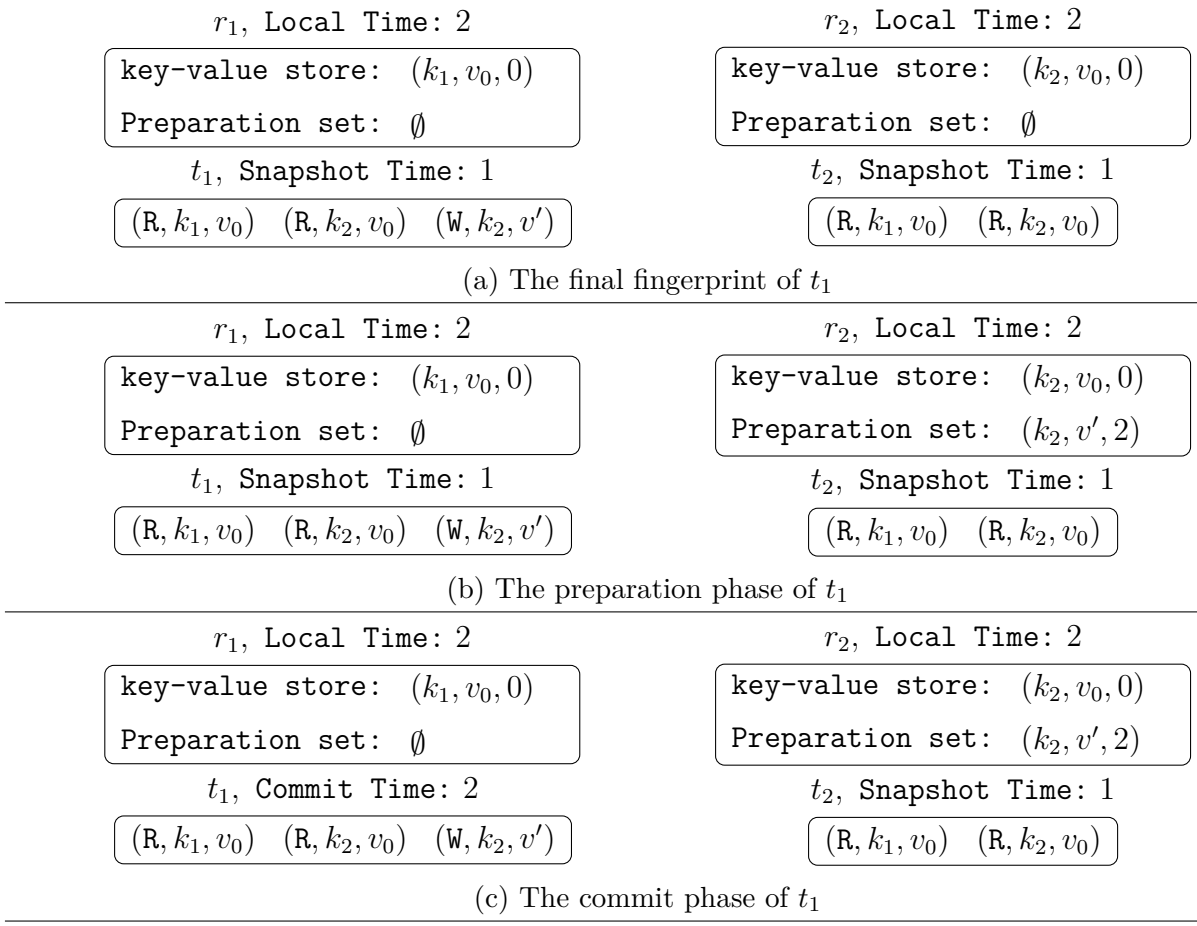
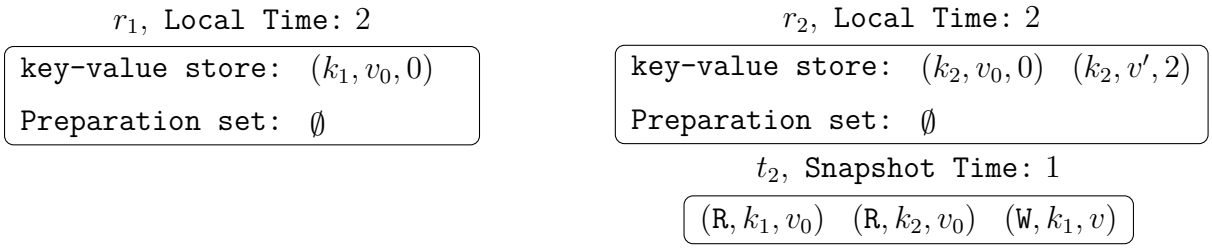


Figure 6.8: Clock-SI two-phase commit protocol

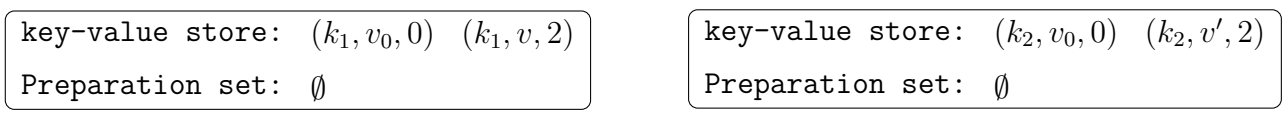
replies with the preparation time of the new versions. Otherwise, the shard rejects the new version due to conflicting writes; and the coordination might abort or restart the transaction.

In the second phase, the coordinator picks the maximum time over all preparation times as the commit time of this transaction. For example in Fig. 6.8c, the commit time of t_1 is 2. The coordinator finally sends the commit time to all shards that contain prepared versions for this transaction. Upon receiving the commit time n, the shard first updates its local time to n, if the local time is behind n. The shard updates the time of prepared versions for this transaction to be n and commits them to the local key-value store. For example, Fig. 6.9a shows the resulting state after t_1. Note that, despite that there is a new version for k_2, this new version does not affect t_2 because: (1) the new version commits after $\mathsf{STimeOf}(t_2)$, hence it does not affect any read of t_2; and (2) t_2 does not update k_2, hence t_2 can commit successfully. An example result of $\mathtt{P}_{\mathtt{Clock}}$ is given in Fig. 6.9b, which exhibits the write skew anomaly.

Du et al. [2013] informally argued that Clock-SI implements snapshot isolation. First, a transaction t carrying a snapshot time $\mathsf{STimeOf}(t)$ read the latest values of keys. Note that the two-phase commits protocol guarantees that when the transaction reads from a shard, there must be no version that can commit before $\mathsf{STimeOf}(t)$. Second, the transaction commits its final effect via the two-phase commits protocol, which guarantees all the versions commit at



(a) Client cl_1 commits t_1 to r_1, while t_2 are still running



(b) Client cl_2 commits t_2 to r_2

Figure 6.9: An example result of t_1 and t_2

the same time to appropriate shards. Recall that the commit time is the maximum time over all preparation times. This means if there is no conflict write between the snapshot and the preparation time in a shard, then it must be no conflict write until the commit time, because any prepared version blocks further update to the key.

6.2.2 Machine States

A shard in Clock-SI comprises a local time and all the history versions of a partition of keys. Each version consists of a value, a time when the version committed and a version state, which is `prepared` or `committed`. In the semantics of Clock-SI, the shards do not explicitly have the preparation set, instead, annotates versions with their states.

Definition 6.21 (Clock-SI local times and versions)**.** *The set of* Clock-SI local times *is defined by:* $\textsc{ClockTime} \stackrel{def}{=} \mathbb{N}$. *Assume a set of* version states $\textsc{ClockState} \stackrel{def}{=} \{\mathtt{prepared}, \mathtt{committed}\}$. *Given the set of values (Def. 4.3), the set of* Clock-SI versions, $\textsc{ClockVersions} \ni \tilde{\nu}$, *is defined by:* $\textsc{ClockVersions} \stackrel{def}{=} \textsc{Value} \times \textsc{ClockTime} \times \textsc{ClockState}$. *Given a version* $\tilde{\nu}$, *let* $\mathsf{ValueOf}(\tilde{\nu})$, $\mathsf{TimeOf}(\tilde{\nu})$ *and* $\mathsf{StateOf}(\tilde{\nu})$ *return the first, second and third projections of* $\tilde{\nu}$ *respectively. Given two versions* $\tilde{\nu}, \tilde{\nu}'$, *the order between them, written* $\tilde{\nu} \sqsubseteq \tilde{\nu}'$, *is defined by:*

$$(v, n, l) \sqsubseteq (v, n', l') \stackrel{def}{\Leftrightarrow} (n = n' \wedge l = l') \vee (l = \mathtt{prepared} \wedge l' = \mathtt{committed} \wedge n \leq n').$$

If two versions $\tilde{\nu} \sqsubseteq \tilde{\nu}'$, then they are either the same version, or $\tilde{\nu}'$ is newer than $\tilde{\nu}$ in the two-phase commit protocol, that is, $\tilde{\nu}'$ is confirmed to commit and but $\tilde{\nu}$ is just prepared. The *local key-value store*, $\tilde{\mathcal{K}}$, of a shard contains all the history versions, which is a *partial function* from keys to lists of Clock-SI versions.

Definition 6.22 (Clock-SI key-value stores)**.** *The set of* Clock-SI local key-value stores *or*

Clock-SI local store *is defined by:*

$$\textsc{ClockKVS} \stackrel{def}{=} \left\{ \tilde{\mathcal{K}} \in \textsc{Key} \rightharpoonup [\textsc{ClockVersions}] \;\middle|\; \texttt{WfClockSIKvs}\left(\tilde{\mathcal{K}}\right) \right\},$$

where `WfClockSIKvs` *is defined by: for any key* $k \in \mathsf{Dom}\left(\tilde{\mathcal{K}}\right)$ *and indices* i, i'*,*

$$\tilde{\mathcal{K}}\,(k, 0) = (v_0, \tilde{t_0}, \texttt{committed}), \tag{6.7}$$

$$\forall v, v' \in \textsc{Value}.\, \forall n, n' \in \textsc{ClockTime}.\, \forall l, l' \in \textsc{ClockState}.$$
$$\tilde{\mathcal{K}}\,(k, i) = (v, n, l) \wedge \tilde{\mathcal{K}}\,(k, i') = (v', n', l') \wedge i < i' \Rightarrow n < n'. \tag{6.8}$$

The set of initial Clock-SI key-value stores*,* $\textsc{ClockKVS}_0 \ni \tilde{\mathcal{K}_0}$*, is defined by:*

$$\textsc{ClockKVS}_0 \stackrel{def}{=} \left\{ \tilde{\mathcal{K}_0} \;\middle|\; \forall k \in \mathsf{Dom}\left(\tilde{\mathcal{K}_0}\right).\, \tilde{\mathcal{K}_0}\,(k) = [(v_0, 0, \texttt{committed})] \right\}.$$

The order $\tilde{\mathcal{K}} \sqsubseteq \tilde{\mathcal{K}'}$ *is defined by point-wise lifting:*

$$\tilde{\mathcal{K}} \sqsubseteq \tilde{\mathcal{K}'} \stackrel{def}{\Leftrightarrow} \forall k \in \textsc{Key}.\, \forall i \in \mathbb{N}.\, \tilde{\mathcal{K}}\,(k, i) \sqsubseteq \tilde{\mathcal{K}'}(k, i).$$

In a well-formed a Clock-SI key-value store, the first version of each key *key* is the initial version, with the initial value v_0 and time 0, and the versions for *key* are ordered over their commit times, captured by the well-formed condition Eq. (6.8). Given the order between versions defined in Def. 6.21, the order between Clock-SI key-value store is defined pointwise.

In a Clock-SI database, versions associated with a key are stored in a distinct shard. This is captured by the definition of Clock-SI machine states, $\mathcal{R}$, comprising a function mapping shard identifiers, r, to their key-value stores and shard local times. Recall that the diffidence between shard local times are bound, however, we do not put any constraint here, because we focus on the correctness of Clock-SI, while the upper bound guarantees progress, which is out of the scoop.

Definition 6.23 (Clock-SI machine states)**.** *Assume a set of* shards identifiers*,* $\textsc{ClockShard} \ni r$*. The set of* Clock-SI machine states*,* $\textsc{ClockSI} \ni \mathcal{R}$*, is defined by:*

$$\textsc{ClockSI} \stackrel{def}{=} \left\{ \begin{array}{l} \mathcal{R} \in \textsc{ClockShard} \stackrel{fin}{\rightharpoonup} \\ \quad \textsc{ClockKVS} \times \textsc{ClockTime} \end{array} \;\middle|\; \begin{array}{l} \forall r, r' \in \mathsf{Dom}\,(\mathcal{R}).\, \forall \tilde{\mathcal{K}}, \tilde{\mathcal{K}'} \in \textsc{ClockKVS}. \\ r \neq r' \wedge (\tilde{\mathcal{K}}, \_) = \mathcal{R}\,(r) \wedge (\tilde{\mathcal{K}'}, \_) = \mathcal{R}\,(r') \\ \qquad \Rightarrow \mathsf{Dom}\left(\tilde{\mathcal{K}}\right) \cap \mathsf{Dom}\left(\tilde{\mathcal{K}'}\right) = \emptyset \end{array} \right\}.$$

The set of initial Clock-SI machine states*,* $\textsc{ClockSI}_0 \ni \mathcal{R}_0$*, is defined by:*

$$\textsc{ClockSI}_0 \stackrel{def}{=} \left\{ \mathcal{R}_0 \;\middle|\; \forall r \in \mathsf{Dom}\,(\mathcal{R}_0).\, \exists \tilde{\mathcal{K}_0} \in \textsc{ClockKVS}_0.\, \mathcal{R}_0\,(r) = \left(\tilde{\mathcal{K}_0}, 0\right) \right\}.$$

Given $\mathcal{R}$*, let* $\mathsf{ShardOf}_{\mathcal{R}}\,(k)$ *denote the shard containing the key* k*. Let* $\mathcal{R}\,(k)$ *denote the list*

of versions associated with the key k, defined by: $\mathcal{R}(k) \stackrel{def}{=} \mathcal{R}(\mathsf{ShardOf}_{\mathcal{R}}(k))_{|0}(k)$. *The order,* $\mathcal{R} \sqsubseteq \mathcal{R}'$, *is defined by:*

$$\forall r \in \mathsf{Dom}(\mathcal{R}).\, \forall \tilde{\mathcal{K}}, \tilde{\mathcal{K}}' \in \textsc{ClockKVS}.\, \forall n, n' \in \textsc{ClockTime}.\\ \mathcal{R}(r) = (\tilde{\mathcal{K}}, n) \Rightarrow \mathcal{R}'(r) = (\tilde{\mathcal{K}}', n') \wedge \tilde{\mathcal{K}} \sqsubseteq \tilde{\mathcal{K}}' \wedge n \leq n'.$$

As explained in Section 6.2.1, transactions in a client session are executed sequentially, yet they might commit to different shards. To handle that, each client maintains a local time, initially being zero, that determines the session order between transactions from this client. The Clock-SI client environments tracks all client local times.

Definition 6.24 (Clock-SI client environments)**.** *Given the set of client identifiers defined in Def. 4.1, the set of* Clock-SI client environments, $\textsc{ClockClientEnv} \ni \tilde{\mathcal{U}}$, *is defined by:*

$$\textsc{ClockClientEnv} \stackrel{def}{=} \left\{ \tilde{\mathcal{U}} \in \mathit{CID} \stackrel{fin}{\rightharpoonup} \textsc{ClockTime} \right\}.$$

The set of initial Clock-SI client environments, $\textsc{ClockClientEnv}_0 \ni \tilde{\mathcal{E}}_0$, *is defined by:*

$$\textsc{ClockClientEnv}_0 \stackrel{def}{=} \left\{ \tilde{\mathcal{E}}_0 \;\middle|\; \forall cl \in \mathsf{Dom}\left(\tilde{\mathcal{E}}_0\right).\, \tilde{\mathcal{E}}_0(cl) = 0 \right\}.$$

Given two Clock-SI environments $\tilde{\mathcal{U}}$,$\tilde{\mathcal{U}}'$, *the order,* $\tilde{\mathcal{U}} \sqsubseteq \tilde{\mathcal{U}}'$, *is defined by:*

$$\tilde{\mathcal{U}} \sqsubseteq \tilde{\mathcal{U}}' \stackrel{def}{\Leftrightarrow} \forall cl \in \mathsf{Dom}\left(\tilde{\mathcal{U}}\right).\, \tilde{\mathcal{U}}(cl) \leq \tilde{\mathcal{U}}'(cl).$$

6.2.3 Reference Implementation and Reference Semantics

We now give a formal operational semantics to Clock-SI. The semantics models the fine-grained execution described in Section 6.2.1, including:

(1) transaction initialisation, when the coordinator sets up the transaction start time;

(2) fine-grained single read and write operations inside transactions; and

(3) the two-phase commit protocol for committing new values.

For brevity, we use the syntax to track the intermediate state of transactions.

Definition 6.25 (Clock-SI runtime commands)**.** *The set of* preparation buffers, $\textsc{ClockBuffer} \ni \mathcal{B}$, *is defined by* $\textsc{ClockBuffer} \stackrel{def}{=} [\textsc{Key} \times \mathbb{N} \times \textsc{ClockTime}]$. *Given the set of commands,* $\mathtt{C}$ *(Def. 4.11), the set of* Clock-SI runtime commands, $\textsc{ClockRunCommands} \ni \mathtt{R}$, *is defined:*

$$\mathtt{R} ::= [\ \mathtt{T}\ ]^{\mathcal{F},\mathcal{B}}_{n,r} \mid \mathtt{C} \mid \mathtt{R}\,;\mathtt{C}$$

for some fingerprint $\mathcal{F}$, Clock-SI time n and shard r. The set of Clock-SI runtime programs, $\textsc{ClockRunProgs} \ni \mathtt{I}$, *is defined by:* $\textsc{ClockRunProgs} \stackrel{def}{=} \mathit{CID} \stackrel{fin}{\rightharpoonup} \textsc{ClockRunCommands}$.

Each runtime transaction, $[\ \mathtt{T}\ ]^{\mathcal{F},\mathcal{B}}_{n,r}$, has the following phases:

(1) the coordinator executes the command of the transaction `T`, reading keys locally or remotely and accumulating the effect to the read-write set $\mathcal{F}$;

(2) when `T` is reduced to `skip`, the coordinator sends the new values of keys in $\mathcal{F}$ to the shards R that contain these keys, and collects the replies that contain receiving times, storing them in $\mathcal{B}$; and

(3) upon receiving all replies, the coordinator picks the maximum time in $\mathcal{B}$ as the actual commit time, and sends the time to R, hence committing the transaction.

Note that, in the Clock-SI protocol, a shard *clockshard* will reject new values of keys, hence the entire transaction t, if there are conflicting writes to these keys: this can happen in the step (2), where *clockshard* rejects the new version because there is another value already either in the key-store or in the committing buffer, whose prepared or committed time is after the transaction start time of t. If this happens the coordinator of t will restart the transaction with a newer transaction start time. However, this restart is not observable by clients. Hence, our operational semantics of Clock-SI focuses on successful transactions. For the purpose of swapping operations in Section 6.2.4, steps in the operational semantics of Clock-SI are explicitly annotated with labels.

Definition 6.26 (Clock-SI semantics labels)**.** *The set of* Clock-SI semantics labels *is defined by:*

$$\textsc{ClockLabels} \stackrel{def}{=} \bigcup_{\substack{cl\in \mathit{CID}, r\in \textsc{ClockShard},\\ l\in\{\mathtt{W},\mathtt{R},\mathtt{P}\}, k\in \mathit{Key},\\ v\in \mathit{Value}, n\in \textsc{ClockTime}}} \left\{ \begin{array}{r} (cl, r, n, \mathtt{S})\,, (cl, r, (l, k, v), n)\,, \\ (cl, r, n, \mathtt{E})\,, (r, n) \end{array} \right\}$$

The snapshot label, $(cl, r, n, \mathtt{S})$, states that the client cl starts a transaction at time n in the coordinator r. The label of the form $(cl, r, (l, k, v), n)$ means a step in internal execution $l = \mathtt{W}$ or $l = \mathtt{R}$, or a step in the first phase (preparation phase) of two phase commit protocol: a transaction with client cl and coordinator r writes $l = \mathtt{W}$, reads $l = \mathtt{R}$ or prepares $l = \mathtt{P}$ key k with value v. For write and read label, the time n is the snapshot time of the transaction. For preparation label, the time n is the preparation time. Label $(cl, r, n, \mathtt{E})$ corresponds the final commit transition of a transaction. Label (r, n) corresponds to a time tick transition in which the shard r advances its local time.

```
1 transSnapshot( trans, clientTime )
2     wait until clientTime < getShardClockTime();
3     trans.snapshotTime = getShardClockTime();
4     trans.read-write-set = ∅;
5     trans.state = active;
```

(a) Reference implementation of `transSnapshot`

$$\textsc{CSISnapshot}\quad \frac{\#\mathbf{2} : \mathcal{R}(r) = (\_, n') \qquad \#\mathbf{2} : n' > n}{(\mathcal{R}, n, s), [\ \mathtt{T}\ ] \xrightarrow{(cl,r,n',\mathtt{S})} (\mathcal{R}, n, s), [\ \mathtt{T}\ ]^{\emptyset,\emptyset}_{n',r}}$$

(b) Reference semantics for `transSnapshot`, where $\#\mathbf{n}$ denotes line n in Fig. 6.10a

Figure 6.10: Clock-SI: transaction start

Transaction initialisation As explained in Section 6.2.1, a client commits a transaction to an *arbitrary* shard and then awaits the confirmation. Upon receiving the request, the shard acts as the *coordinator* of the transaction in that it is responsible for fetching values of keys locally or remotely, and committing any new values to the appropriate shards using a two-phase commits protocol. The coordinator initialises the transaction snapshot time, by calling `transSnapshot`. The reference implementation for `transSnapshot` is given in Fig. 6.10:

(1) the coordinator assigns the shard local time as the *snapshot time* for this transaction, `trans`, if the coordinator local time is greater than the minimum transaction snapshot time, `clientTime`, (line 2 in Fig. 6.10a), otherwise, the shard postpones the transaction;

(2) the coordinator initialises the read-write set, `read-write-set`, of the transaction as the empty set; and

(3) transaction state is set to be `active`.

Rule CSISnapshot in Fig. 6.10b captures the semantics of `transSnapshot`. The resulting runtime transaction, of the form $[\ \mathtt{T}\ ]^{\mathcal{F},\mathcal{B}}_{n,r}$, tracks the runtime information of this transaction, tracking the snapshot time n, a the coordinator r, the initial fingerprint $\mathcal{F} = \emptyset$ and the *preparation buffer* $\mathcal{B} = \emptyset$.

Write operation We now explain how the coordinator executes the internal command of a transaction. For any mutation command of the form $\mathtt{wt}\ (k, \mathtt{E})$, the coordinator calls `transWrite` defined in Fig. 6.11b. The function erases any previous write entry of the key and adds the new value of the key to the read-write set. Rule CSIWrite in Fig. 6.12b captures the semantics of this function. The new write operation is added into $\mathcal{F}$ via $\lessdot$. Recall that the fingerprint combination operation $\ll$ is defined in Def. 4.16.

```
1 transWrite( trans, k, v )
2     trans.read-write-set = trans.read-write-set \ (w,k,-);
3     trans.read-write-set = trans.read-write-set + (w,k,v);
```

(a) Reference implementation of `transWrite`

CSIWrite

$$\frac{k = [\![\mathtt{E}_1]\!]_s \qquad v = [\![\mathtt{E}_2]\!]_s}{(\mathcal{R}, n, s)\,,\, [\ \mathtt{wt}\,(\mathtt{E}_1, \mathtt{E}_2)\,;\mathtt{T}\ ]^{\mathcal{F},\emptyset}_{n',r} \xrightarrow{(cl,r,(\mathtt{W},k,v),n')} (\mathcal{R}, n, s)\,,\, [\ \mathtt{T}\ ]^{\mathcal{F} \lessdot (\mathtt{W},k,v),\emptyset}_{n',r}}$$

(b) Reference semantics of `transWrite`

Figure 6.11: Clock-SI: transactional write

```
1 transRead( trans, k )
2     if ( (w,k,v) in trans.read-write-set ) return v;
3     else if ( (r,k,v) in trans.read-write-set ) return v;
4     else {
5         v = readFromShard(shard, trans.snapshotTime);
6         trans.read-write-set = trans.read-write-set + (r,k,v);
7         return v;
8     }
9
10 readFromShard(shard, snapshotTime, k)
11     wait snapshotTime < getShardClockTime();
12     i = 0;
13     for ver of getStore()
14         if (ver.key = k and ver.time <= snapshotTime) {
15             wait until ver.state == committed;
16             if(i <= ver.time) i = ver.time;
17         }
18     return getValue(k,i);
```

(a) Reference implementation of `transRead`

CSIReadL

$$\frac{k = [\![\mathtt{E}]\!]_s \qquad \#\mathbf{2}, \#\mathbf{3} : (\mathtt{W}, k, v) \in \mathcal{F} \vee ((\mathtt{W}, k, v) \notin \mathcal{F} \wedge (\mathtt{R}, k, v) \in \mathcal{F})}{(\mathcal{R}, n, s)\,,\, [\ \mathtt{x} := \mathtt{rd}\,(\mathtt{E})\,;\mathtt{T}\ ]^{\mathcal{F},\emptyset}_{n',r} \xrightarrow{(cl,r,(\mathtt{R},k,v),n')} (\mathcal{R}, n, s\,[\mathtt{x} \mapsto v])\,,\, [\ \mathtt{T}\ ]^{\mathcal{F},\emptyset}_{n',r}}$$

CSIReadS

$$\frac{\begin{array}{c} k = [\![\mathtt{E}]\!]_s \qquad \forall l, v.\,(l, k, v) \notin \mathcal{F} \qquad \#\mathbf{5} : r' = \mathsf{ShardOf}_{\mathcal{R}}\,(k) \qquad (\tilde{\mathcal{K}}, n'') = \mathcal{R}\,(r') \\ \#\mathbf{11} : n'' > n' \qquad \#\mathbf{13}, \#\mathbf{14} : \tilde{\mathcal{V}} = \left\{ \tilde{\nu}' \;\middle|\; \exists i.\,\tilde{\mathcal{K}}\,(k, i) = \tilde{\nu}' \wedge \mathsf{TimeOf}\,(\tilde{\nu}') \leq n' \right\} \\ \#\mathbf{15} : \forall \tilde{\nu} \in \tilde{\mathcal{V}}.\,\mathsf{StateOf}\,(\tilde{\nu}) = \mathtt{committed} \qquad \#\mathbf{16} : (k, v, \_) = \mathsf{Max}\left(\tilde{\mathcal{V}}\right) \end{array}}{(\mathcal{R}, n, s)\,,\, [\ \mathtt{x} := \mathtt{rd}\,(\mathtt{E})\,;\mathtt{T}\ ]^{\mathcal{F},\emptyset}_{n',r} \xrightarrow{(cl,r',(\mathtt{R},k,v),n')} (\mathcal{R}, n, s\,[\mathtt{x} \mapsto v])\,,\, [\ \mathtt{T}\ ]^{\mathcal{F} \lessdot (\mathtt{R},k,v),\emptyset}_{n',r}}$$

(b) Reference semantics of `transRead`, where #**n** denotes line n in Fig. 6.12a

Figure 6.12: Clock-SI: transactional read

Read operation For any look-up command of the form `x := rd(E)`, the coordinator calls `transRead` defined in Fig. 6.12a. If the key has been updated or read in the transaction, the coordinator directly reads from the read-write set of the transaction, that is, lines 2 and 3 respectively in Fig. 6.12a. Rule CSIReadL in Fig. 6.12b captures the semantics of a local read. Otherwise, the transaction reads from the shard that contains the key via function call `readFromShard` defined in Fig. 6.12a. Once receiving the read request, the remote shard, first, will wait until the local time is greater than the transaction snapshot time (line 11), and then wait until all the versions with time-stamps smaller than the snapshot time are committed successfully (line 15). If both conditions are satisfied, the transaction replies with the latest version before the snapshot time (lines 16 and 18). The coordinator then will add the version into the read-write set (line 6). Rule CSIReadS rule in Fig. 6.12b captures the semantics or remote read. The premise $n'' > n'$ states that the snapshot time precedes the local shard time. The variable $\tilde{\mathcal{V}}$ captures all the versions, either in the prepared or committed states, that has smaller timestamp than the transaction snapshot time, n'. Recall that, it must also include the prepared versions, because these versions may turn to committed versions in the future. The premise $\mathsf{StateOf}(\tilde{\nu}) = \texttt{committed}$ states that versions $\tilde{\nu} \in \tilde{\mathcal{V}}$ have committed successfully, hence it is safe to read, because the list cannot change any more. The last premise fetches the version with the maximum time-stamp in $\tilde{\mathcal{V}}$, that is, $(k, v, \_) = \mathsf{Max}\left(\tilde{\mathcal{V}}\right)$. This version is then added into the fingerprint $\mathcal{F}$. Note that, both read and write steps do not change the client local time.

Two-phase commit protocol: prepared phase When the transactional command reaches the end, that is, `skip`, the coordinator commits all the write operations in the final read-write set to the appropriate shards using a two-phase commit protocol. The reference implementation is given in Fig. 6.13a. In the first phase, for every write operation, `(w,k,v)`, in the write-read set, `trans.read-write-set`, the coordinator asynchronously sends the new value `v`, and the snapshot time `snapshotTime` of the transaction, that is, a message of the form "`prepare` (k, v, n)" to the shard that contains `k` (line 5), and awaiting a reply (line 6). Upon receiving the preparation request, the remote shard checks if there is a concurrent write since the snapshot time `snapshotTime`. If there is no conflicting write (line 18), the shard assigns the shard local time as the *prepared time* (line 19), adds the new value `v` of `k` together with the prepared time to the preparation set (line 20), and echoes the message but modifies the time to be the prepared time (line 21). Note that versions in the preparation set are visible to others. If a transaction t wants to read a key, and if there is a prepared version of the key for which the preparation time is smaller than the snapshot time of the transaction, the transaction t must wait until this entry has been committed or aborted. Also, prepared versions will stop other transactions committing to the same key.

Rule CSIPrepare in Fig. 6.13b states that the shard r receives a new value v of a key k. The premise $n' < n''$ requires that the local time of the shard r be greater than the snapshot

```
1 commitTrans( trans )
2     prep = [];
3     snapshotTime = trans.snapshotTime;
4     for (w,k,v) in trans.read-write-set {
5         send ``prepare (k,v,snapshotTime)'' to shardOf(k);
6         wait ``(k,v,t)'';
7         prep = prep + (k,v,t);
8     }
9
10    commitTime = maxTime(prep)
11
12    for (k,v,t) in prep {
13        send ``commit (k,v,commitTime)'' to shardOf(k);
14    }
15
16 On receive ``prepare (k,v,snapshotTime)''
17    wait snapshotTime < getShardClockTime();
18    if noConcurrentWriteSince(k,snapshotTime) {
19        preparedTime = getShardClockTime();
20        log ``(k,v,preparedTime,prepared)'' in preparation set;
21        send ``(k,v,preparedTime)'';
22    } else {
23        // restart this transaction entirely.
24        ...
25    }
26
27 On receive ``commit (k,v,commitTime)''
28    insert(k,v,commitTime,committed);
```

(a) Reference implementation of `commitTrans`

CSIPrepare

$$\frac{\#\mathbf{4}: (\mathtt{W},k,v) \in \mathcal{F} \qquad \#\mathbf{5}: r = \mathsf{ShardOf}_{\mathcal{R}}(k) \qquad (\tilde{\mathcal{K}}, n'') = \mathcal{R}(r)}{\begin{array}{c}\#\mathbf{17}: n' < n'' \qquad \#\mathbf{18}: \tilde{\mathcal{V}} = \tilde{\mathcal{K}}(k) \qquad \#\mathbf{18}: \forall i.\, 0 \le i < \left|\tilde{\mathcal{V}}\right| \Rightarrow \mathsf{TimeOf}\left(\tilde{\mathcal{V}}(i)\right) < n' \\ \#\mathbf{20}: \mathcal{R}' = \mathcal{R}\left[r \mapsto \left(\tilde{\mathcal{V}} :: [(v, n'', \mathtt{prepared})], n''\right)\right]\end{array}}{(\mathcal{R}, n, s), [\ \mathtt{skip}\ ]^{\mathcal{F},\mathcal{B}}_{n',r} \xrightarrow{(cl,r',(\mathtt{P},k,v),n'')} (\mathcal{R}', n, s), [\ \mathtt{skip}\ ]^{\mathcal{F}\setminus\{(\mathtt{W},k,v)\},\mathcal{B}\uplus\left\{(k,\left|\tilde{\mathcal{V}}\right|,n'')\right\}}_{n',r}}$$

CSICommit

$$\frac{\begin{array}{c}\forall k, v.\, (\mathtt{W},k,v) \notin \mathcal{F} \\ \#\mathbf{10}: n'' = \mathsf{MaxTime}(\mathcal{B}) \qquad \#\mathbf{21}, \#\mathbf{28}: \mathcal{R}'' = \mathsf{CLOCKUpdate}(\mathcal{R}, \mathcal{B}, n'')\end{array}}{(\mathcal{R}, n, s), [\ \mathtt{skip}\ ]^{\mathcal{F},\mathcal{B}}_{n',r} \xrightarrow{(cl,r,n'',\mathtt{E})} (\mathcal{R}'', n'', s), \mathtt{skip}}$$

(b) Reference semantics of `commitTrans`, where #**n** denotes line n in Fig. 6.13a

Figure 6.13: Clock-SI: transaction commit

time n'. The second last premise, $\mathsf{TimeOf}\left(\tilde{\mathcal{V}}(i)\right) < n'$, states that no conflicting write: that is, existing versions of key k must have smaller time-stamp than the snapshot time n'. In our semantics of Clock-SI, we do not explicitly have preparation set, but labels versions with their states. Hence, we add the version into the store and labelled them with the preparation time n''

and state tag `prepared` (the last premise of CSIPREPARE). Finally, this new version is erased from the fingerprint $\mathcal{F}$, and added into the buffer, updating the buffer to $\mathcal{B} \uplus \left\{(k, \left|\tilde{\mathcal{V}}\right|, n'')\right\}$. Note that the label of this transition tracks the preparation time of this version, instead of the snapshot time.

Two-phase commit protocol: commit phase In the second phase of commit, the coordinator collects all the replies (line 7), assigns the final commit time `commitTime` of this transaction as the maximum of all the preparation times (line 10). The coordinator then send a message of the form `commit (k,v,commitTime)` to the shards that contain versions of this transaction. Upon receiving the commit time n', the shard inserts the versions at n' to their local stores (line 28). For brevity, the entire second phase is captures by a commit step transition, rule CSICOMMIT, in the semantics. The actual commit time is given by $\mathsf{MaxTime}$ function in the premise of CSICOMMIT.

Definition 6.27 ($\mathsf{MaxTime}$ function)**.** *Given a set of preparation buffers $\mathcal{B}$, $\mathsf{MaxTime}$ is defined by* $\mathsf{MaxTime}(\mathcal{B}) \stackrel{def}{=} \mathsf{Max}(\mathsf{Times}(\mathcal{B}))$*, where* $\mathsf{Times}(\mathcal{B}) \stackrel{def}{=} \{n \mid (\_, \_, n) \in \mathcal{B}\}$.

The last premise, $\mathsf{CLOCKUpdate}(\mathcal{R}, \mathcal{B}, n'')$, inserts all the preparation versions. This function alters the time-stamp of versions included in the buffer $\mathcal{B}$ to the actual commit time n'', and changes the state tag to `committed`. Function $\mathsf{CLOCKUpdate}(\mathcal{R}, \mathcal{B}, n'')$ is defined pointwise on all the key-value store in $\mathcal{R}$. Given a key k, and the version at i^{th} index, in the preparation buffer, that is $(k, i, \_) \in \mathcal{B}$, and given the commit time n, the auxiliary function $\mathsf{CLOCKUpdateKVS}\left(\tilde{\mathcal{K}}, k, i, n\right)$ update the key-value store $\tilde{\mathcal{K}}$ in the shard that hosts k, updating the state of i^{th} version of k to `committed`, and the time to the commit time n.

Definition 6.28 ($\mathsf{CLOCKUpdate}$ function)**.** *Given a Clock-SI machine state $\mathcal{R}$, a set of preparation buffers $\mathcal{B}$, and a time n,* $\mathsf{CLOCKUpdate}$ *is defined by:*

$$\begin{aligned}
\mathsf{CLOCKUpdate}(\mathcal{R}, \emptyset, n) &\stackrel{def}{=} \mathcal{R} \\
\mathsf{CLOCKUpdate}(\mathcal{B} \uplus \{(k, i, _)\}, n) &\stackrel{def}{=} \texttt{let } r = \mathsf{ShardOf}_{\mathcal{R}}(k), \quad (\tilde{\mathcal{K}}, n') = \mathcal{R}(r), \\
&\qquad \texttt{and } \tilde{\mathcal{K}}' = \mathsf{CLOCKUpdateKVS}\left(\tilde{\mathcal{K}}, k, i, n\right) \texttt{ in} \\
&\qquad \mathsf{CLOCKUpdate}(\mathcal{R}, \mathcal{B})\left[r \mapsto (\tilde{\mathcal{K}}', n')\right]
\end{aligned}$$

where $\mathsf{CLOCKUpdateKVS}$ *is defined by:*

$$\begin{aligned}
\mathsf{CLOCKUpdateKVS}\left(\tilde{\mathcal{K}}, k, i, n\right) \stackrel{def}{=} \; & \texttt{let } \tilde{\nu} = \tilde{\mathcal{K}}(k), \quad (v, _, _) = \tilde{\nu}(n) \\
& \texttt{and } \tilde{\nu}' = \tilde{\nu}\left[i \mapsto (v, n, \texttt{committed})\right] \texttt{ in } \tilde{\mathcal{K}}' = \tilde{\mathcal{K}}'\left[k \mapsto \tilde{\nu}'\right].
\end{aligned}$$

program The operational semantics of Clock-SI is a standard interleaving semantics, depicted in Fig. 6.14, where a client takes a step in turn, captured by CSITRANS. The abstract states of Clock-SI semantics comprise the machine states of Clock-SI and client environments.

$$\textsc{CSITrans}\quad \frac{n = \tilde{\mathcal{U}}\left(cl\right) \qquad s = \mathcal{E}\left(cl\right) \qquad \mathtt{R} = \mathtt{I}(cl) \qquad \iota = (cl, \cdots) \qquad \left(\mathcal{R}, n, s\right), \mathtt{R} \xrightarrow{\iota} \left(\mathcal{R}', n', s'\right), \mathtt{R}'}{\left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}\right), \mathtt{I} \xrightarrow{\iota} \left(\mathcal{R}', \tilde{\mathcal{U}}\left[cl \mapsto n\right], \mathcal{E}\left[cl \mapsto s'\right]\right), \mathtt{I}\left[cl \mapsto \mathtt{R}'\right]}$$

$$\textsc{CSITick}\quad \frac{(\tilde{\mathcal{K}}, n) = \mathcal{R}\left(r\right) \qquad \mathcal{R}' = \mathcal{R}\left[r \mapsto (\tilde{\mathcal{K}}, n + n')\right]}{\left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}\right), \mathtt{I} \xrightarrow{(r, n+n')} \left(\mathcal{R}', \tilde{\mathcal{U}}, \mathcal{E}\right), \mathtt{I}}$$

Figure 6.14: Clock-SI: semantics for programs

Definition 6.29 (Clock-SI configurations)**.** *the set of* Clock-SI configurations*,* $\textsc{ClockConf} \ni \Theta$*, is defined by:* $\textsc{ClockConf} \stackrel{def}{=} \textsc{ClockSI} \times \textsc{ClockClientEnv}$*. and the set of* initial Clock-SI configurations*,* $\textsc{ClockConf}_0 \ni \Theta$*,* $\textsc{ClockConf}_0 \stackrel{def}{=} \textsc{ClockSI}_0 \times \textsc{ClockClientEnv}_0$.

Last, in Clock-SI, the shard local times are the physical times. To model the physical time ticks, a shard can arbitrarily advance its local time, captured by CSITick.

We present the formal, fine-grained operational semantics for Clock-SI. Recall that a commit may fail in the first phase due to conflicting writes, however, we only focus on the correctness of this protocol, hence, our reference operational semantics only models the successful cases. Therefore the set of Clock-SI traces only contains successful traces in that all clients reach the end, `skip`

Definition 6.30 (Clock-SI traces)**.** *The set of* Clock-SI traces*,* $\textsc{ClockTrace} \ni \phi$*, is defined by*

$$\textsc{ClockTrace} \stackrel{def}{=} \bigcup_{n \in \mathbb{N}} \left\{ \phi \;\middle|\; \begin{array}{r} \exists \mathtt{P} \in \textsc{Progs}.\, \phi \in \mathsf{ClockTraceN}\left(n\right) \wedge \mathsf{Last}\left(\phi\right) = ((\_, \_, \_), \mathtt{P}) \\ \wedge\, \forall cl \in \mathsf{Dom}\left(\mathtt{P}\right).\, \mathtt{P}\left(cl\right) = \mathtt{skip} \end{array} \right\}$$

where ClockTraceN *is defined by:*

$$\mathsf{ClockTraceN}\left(0\right) \stackrel{def}{=} \left\{ \left(\mathcal{R}_0, \tilde{\mathcal{E}}_0, \mathcal{E}_0\right), \mathtt{P}_0 \;\middle|\; \begin{array}{r} \mathcal{R}_0 \in \textsc{ClockSI}_0 \wedge \tilde{\mathcal{E}}_0 \in \textsc{ClockClientEnv}_0 \\ \wedge\, \mathtt{P} \in \textsc{Progs} \wedge \mathsf{Dom}\left(\mathtt{P}_0\right) \subseteq \mathsf{Dom}\left(\tilde{\mathcal{E}}_0\right) \cap \mathsf{Dom}\left(\mathcal{E}_0\right) \end{array} \right\}$$

$$\mathsf{ClockTraceN}\left(n+1\right) \stackrel{def}{=} \left\{ \mathsf{Last}\left(\phi\right) \xrightarrow{\iota} \left(\mathcal{R}', \tilde{\mathcal{U}}', \mathcal{E}'\right), \mathtt{I}' \;\middle|\; \phi \in \mathsf{ClockTraceN}\left(n\right) \wedge \iota \in \textsc{ClockLabels} \right\}$$

Two traces, ϕ, ϕ'*, are equivalent, written* $\phi \simeq \phi'$*, if and only if*

$$\begin{array}{r} \forall \mathcal{R}, \mathcal{R}'.\, (\mathcal{R}, \_) = \mathsf{Last}\left(\phi\right) \wedge (\mathcal{R}', \_) = \mathsf{Last}\left(\phi'\right) \Rightarrow \mathsf{Dom}\left(\mathcal{R}\right) = \mathsf{Dom}\left(\mathcal{R}'\right) \\ \wedge\, \forall r \in \mathsf{Dom}\left(\mathcal{R}\right), \tilde{\mathcal{K}}, \tilde{\mathcal{K}}'.\, \left(\mathcal{R}\left(r\right) = (\tilde{\mathcal{K}}, \_) \wedge \mathcal{R}\left(r'\right) = (\tilde{\mathcal{K}}', \_) \Rightarrow \tilde{\mathcal{K}} = \tilde{\mathcal{K}}'\right). \end{array}$$

Two machine states are equivalent if for each shard r, the key-value stores of r have the same

state. Two Clock-SI traces are equivalent if and only if the final machine states of the two traces are equivalent. This notation of *trace equivalence* is important for the verification of Clock-SI protocol: we will show that the fine-grained trace can be converted to an equivalent *normalised* trace that transactions do not interleave in Section 6.2.4. This normalised trace, can be encoded into a kv-store trace in Section 6.2.5, and proven satisfied the execution test for `SI`.

6.2.4 Verification: Annotated Normalised Traces

We now introduce *annotated* Clock-SI traces and *annotated normalised* Clock-SI traces. For the purpose of manipulating a Clock-SI trace, we annotate the read, write and preparation operations in the trace, with the actual commit time of this transaction obtained in the commit phase of the two-phase commit. An annotate normalised trace is a trace where transitions of transactions are not interfered by other transactions. This annotate normalised traces are the key for verifying Clock-SI protocol, because transactions in the traces are executed atomically.

Given a Clock-SI trace, for each transaction, we annotate read, write and preparation steps with the commit time of the transaction. This extra information is useful for swapping these steps. We call the new traces *annotated Clock-SI traces*.

Definition 6.31 (Annotated Clock-SI traces)**.** *Given a Clock-SI trace, an extended Clock-SI trace is defined inductively by:*

$$\mathsf{AnnoClock}\left(\left(\mathcal{R},\tilde{\mathcal{U}},\mathcal{E}\right),\mathtt{P}\right) \stackrel{def}{=} \left(\mathcal{R},\tilde{\mathcal{U}},\mathcal{E}\right),\mathtt{P}$$

$$\mathsf{AnnoClock}\left(\left(\mathcal{R},\tilde{\mathcal{U}},\mathcal{E}\right),\mathtt{P}\xrightarrow{\iota}\phi\right) \stackrel{def}{=} \begin{cases} \left(\mathcal{R},\tilde{\mathcal{U}},\mathcal{E}\right),\mathtt{P}\xrightarrow{\iota}\mathsf{AnnoClock}\left(\phi\right) & \iota\neq(cl,r,(l,k,v),n) \\ \left(\mathcal{R},\tilde{\mathcal{U}},\mathcal{E}\right),\mathtt{P}\xrightarrow{\iota,n'}\mathsf{AnnoClock}\left(\phi\right) & \\ \qquad \iota=(cl,r,(l,k,v),n)\wedge(cl,r,n',\mathtt{E})=\mathsf{FirstCommit}\left(\phi,cl\right) & \end{cases}$$

where $\mathsf{FirstCommit}\left(\phi,cl\right)$ *returns the first commit for client* cl *in the trace* ϕ*. Let* $A\textsc{ClockTrace}$ *denote the set of the extended traces.*

In an annotated Clock-SI trace, a transaction might interfere with others, but in our kv-store trace, a transaction is executed atomically. We now introduce annotated normalised Clock-SI traces in which transactions are executed atomically. We then show that every annotated Clock-SI trace can be transformed into an annotated normalised trace. In the normalised, we separate transactions into two parts:

(1) snapshot step, (r,n'); and

(2) the rest including read, write, preparation, and commit steps.

The first snapshot step is *allowed* to interfered with any other steps, since it has no direct effect. Additionally, the time tick step is also *allowed* to interfere with other steps. This leads us to define a notion of *time and snapshot* segments, $\texttt{TickAndSnapshot}\,(\phi, r)$ these only contain snapshot and time tick steps in shard r.

Definition 6.32 (Time and Snapshot Segments)**.** *Given a shard r, a trace segment ϕ only contains time-tick steps in r or snapshot steps in r, written $\texttt{TickAndSnapshot}\,(\phi, r)$, if and only if*

$$\exists m.\, m = |\phi| \wedge \forall \tilde{\alpha}_0, \cdots, \tilde{\alpha}_{m-1}, \iota_1, \cdots, \iota_{m-1}.\, \phi = \tilde{\alpha}_0 \xrightarrow{\iota_1} \tilde{\alpha}_1 \xrightarrow{\iota_2} \cdots \xrightarrow{\iota_{n-1}} \tilde{\alpha}_{m-1} \\ \Rightarrow \bigwedge_{0<i<m} \exists cl', n'.\, (\iota_i = (r, n') \vee \iota_i = (cl', r, n', \texttt{S}))\,.$$

In a normalised trace $\texttt{ANCLOCKTrace}\,(\phi)$, defined by Eq. (6.9), any read, write or preparation step ι from a transaction from a client cl and its coordinator r is followed by arbitrary numbers of time tick steps or snapshot steps in the coordinator r, captured by `TickAndSnapshot` predicate, and then followed by a step ι' from the same transaction. More specifically, if a step ι is a write (`W`), read (`R`), or preparation (`P`) step from a client cl, and the coordinator shard r, and if this trace is in normalised form, then the following client observable step ι' must be also steps from the cl, that is, ι' is a write (`W`), read (`R`), preparation (`P`), or a commit step (`E`), from the same client cl. We include the commit step as a guard, indicating the end of a transaction. Recall that, both time tick in r, and snapshot step from other client do not directly affect the execution of the current transaction, therefore between ι and ι', arbitrary numbers of these two steps are allowed, captured by `TickAndSnapshot` predicate.

Definition 6.33 (Annotated normalised Clock-SI traces)**.** *An annotated normalised Clock-SI trace, written $\texttt{ANCLOCKTrace}\,(\phi)$, is defined by: for all trace segments ϕ', ϕ'', ϕ''', labels ι, ι', clients cl, cl', shards r, r', keys k, k', values v, v', times n, n'*

$$\phi = \phi' \xrightarrow{\iota} \phi'' \xrightarrow{\iota'} \phi''' \wedge \\ (\iota = (cl, r, (\texttt{W}, k, v)\,, n, n'') \vee \iota = (cl, r, (\texttt{R}, k, v)\,, n, n'') \vee \iota = (cl, r, (\texttt{P}, k, v)\,, n, n'')) \\ \Rightarrow \texttt{TickAndSnapshot}\,(\phi'', r) \wedge \\ \begin{pmatrix} \iota' = (cl, r, (\texttt{W}, k', v')\,, n, n'') \vee \iota' = (cl, r, (\texttt{R}, k', v')\,, n, n'') \\ \vee\, \iota' = (cl, r, (\texttt{P}, k', v')\,, n', n'') \vee \iota' = (cl, r, n'', \texttt{E}) \end{pmatrix}. \quad (6.9)$$

In Clock-SI, there two important times, snapshot time and preparation time, which, the former decides the fingerprint of the transaction, and the latter decides if the transaction is allowed (no conflict write) to commit. Recall that rule CSIReadS in Fig. 6.12b fetches the lastest values prior the snapshot time. Rule CSIPrepare in Fig. 6.13b check if all existing versions are committed or prepared before the preparation time of the current transaction. This means, given a transaction t any internal read and write steps can be swap to the right, delaying these

steps in the trace. and any preparation and commit steps can be swap to the left, executing earlier these steps in the trace. Note that if we swap a preparation or commit step with a time tick step, the preparation or commit time may be different, hence, when we swap these two steps, we should also carry the time tick together. The formal claim is in Theorem 6.34.

Theorem 6.34 (Clock-SI equivalent normalised traces)**.** *For any annotated Clock-SI trace ϕ, there exists an equivalent normalised trace ϕ_1, that is, $\phi \simeq \phi_1$ and* $\texttt{ANCLOCKTrace}\,(\phi_1)$.

Proof. Initially let $\phi_1 = \phi$. We then preform the following transformation on the trace ϕ_1 until it satisfies $\texttt{ANCLOCKTrace}\,(\phi_1)$.

(1) **Left mover: Clock-SI preparation and commit steps.** Given the trace ϕ_1, take the transaction t with the earliest commit time such that: a preparation step of this transaction, $\iota_1 = (cl, r, (\texttt{P}, k, v), n', n)$, and the next preparation step or commit step of this transaction, ι' have been interfered by steps from other clients. This means that:

$$\begin{aligned}
\phi_1 = \phi_2 \xrightarrow{\iota_2} \phi' \xrightarrow{\iota''} \left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}\right), \texttt{I} \xrightarrow{\iota} \phi''' \xrightarrow{\iota'} \phi'' \\
\wedge\, \texttt{TickAndSnapshot}\,(\phi''', r) \\
\wedge\, (\iota' = (cl, r, (\texttt{P}, k, v), n'', n) \vee \iota' = (cl, r, n, \texttt{E})) \\
\wedge\, \exists cl', r', r'', l', k', v', n', n_1.\, cl \neq cl' \wedge r \neq r'' \\
\wedge \begin{pmatrix} \iota = (cl', r', (l', k', v'), n', n_1) \vee \iota = (cl', r'', n', \texttt{S}) \\ \vee\, \iota = (cl', r', n', \texttt{E}) \vee \iota = (r'', n') \end{pmatrix}
\end{aligned} \tag{6.10}$$

where ϕ' may also contain steps of transaction. However, let first focus on the step ι. Note that since we always pick the first transaction with labels ι_1, ι' of the form in Eq. (6.10). Therefore, if ι is a read, write or preparation step, or a commit step from other client, the transaction of ι must commit after n', that is, $n_1 \geq n$. By Eq. (6.10), $n_1 \geq n$ and Prop. 6.36, it allows use to move ι to the left: for some trace segment ϕ_3 modified from ϕ''' and ι_2 modified from ι:

$$\left(\phi_2 \xrightarrow{\iota_1} \phi' \xrightarrow{\iota''} \phi_3 \xrightarrow{\iota'} \_ \xrightarrow{\iota_2} \phi''\right) \simeq \phi_1$$

We assign the new trace to ϕ_1 and go back to the beginning of step (1). This process must terminate because each iteration the labels ι and ι' moves closer to each other and there are only finite transactions in the trace. In the end, the trace ϕ_1 satisfies: (i) the preparation steps and commit step for each transaction are not interfered by other clients; and (ii) transactions commit in the order of their commit times. This means that: for all trace segments ϕ', ϕ'', ϕ''', labels ι, ι', client cl, cl', shard r, r', keys k, k', values v, v', times

n, n', n''

$$\begin{aligned}\phi_1 = \phi' \xrightarrow{\iota} \phi'' \xrightarrow{\iota'} \phi''' \wedge \iota = (cl, r, (\mathtt{P}, k, v), n, n'') \\ \Rightarrow \mathtt{TickAndSnapshot}\,(\phi'', r) \wedge (\iota = (cl, r, (\mathtt{P}, k', v'), n') \vee \iota = (cl, r, n', \mathtt{E})) \\ \wedge \forall \iota'' \in \phi'''. \forall r', cl', k'', v'', n''', n_1. (\iota'' = (cl', r', (\mathtt{P}, k'', v''), n''', n_1) \Rightarrow n_1 \geq n'').\end{aligned} \tag{6.11}$$

(2) **Right mover: Clock-SI internal read and write steps.** Given an annotated trace ϕ_1 that satisfies Eq. (6.11). We now move all the internal reads and writes of a transaction towards the preparation and commit of the transaction. Given the trace ϕ_1, we pick a read or write ι such that: for some $\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}, \mathtt{I}$,

$$\begin{aligned}\phi_1 = \phi' \xrightarrow{\iota} \left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}\right), \mathtt{I} \xrightarrow{\iota'} \phi'' \wedge (\iota = (cl, r, (\mathtt{W}, k, v), n, n') \vee \iota = (cl, r, (\mathtt{R}, k, v), n, n')) \\ \wedge \forall l, k', v', n''. \iota' \neq (cl, r, (l, k', v'), n'') \wedge \iota' \neq (cl, r, n', \mathtt{E})\end{aligned}$$

By Prop. 6.35, we can move the ι to the right, which yields

$$\left(\phi' \xrightarrow{\iota'} \left(\mathcal{R}', \tilde{\mathcal{U}}'\right), \mathcal{E}' \xrightarrow{\mathtt{I}'} \phi''\right) \simeq \phi_1$$

for some new $\mathcal{R}', \tilde{\mathcal{U}}', \mathcal{E}', \mathtt{I}'$. We assign the new trace to ϕ_1 and go back to the beginning of step (2). This process must terminate because each iteration the labels ι of a transaction moves closer to the first preparation step or commit step of the same transaction, and there are only finite transactions in the trace. In the end, the trace ϕ_1 is a normalised trace, that is, $\mathtt{ANCLOCKTrace}\,(\phi_1)$. ■

Proposition 6.35 (Right mover: Clock-SI internal read and write steps)**.** *Assume a Clock-SI trace $\phi \in A\textsc{Clock}\textsc{Trace}$, and two adjacent transitions with labels ι, ι' such that*

$$\begin{aligned}\phi = \phi' \xrightarrow{\iota''} \left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}\right), \mathtt{I} \xrightarrow{\iota} \left(\mathcal{R}', \tilde{\mathcal{U}}', \mathcal{E}'\right), \mathtt{I}' \xrightarrow{\iota'} \phi'' \\ \wedge (\iota = (cl, r, (\mathtt{W}, k, v), n, n'') \vee \iota = (cl, r, (\mathtt{R}, k, v), n, n'')) \\ \wedge \forall l, k', v', n', n'''. \iota' \neq (cl, r, (l, k', v'), n', n''') \wedge \iota' \neq (cl, r, n''', \mathtt{E})\end{aligned}$$

where the rest free variables are universally quantified. The transition labelled by ι can be moved to the right, that is, there exists a new equivalent trace ϕ_1 such that

$$\phi_1 = \phi' \xrightarrow{\iota''} \_ \xrightarrow{\iota'} \_ \xrightarrow{\iota} \phi'' \wedge \phi \simeq \phi_1.$$

Proof sketch. By CSIWrite, any write step does not depend on the state of Clock-SI database. By CSIReadL, any read step that reads from the local state of a transaction does not depend on the state of Clock-SI database. Therefore, for these two cases, it is trivial that we can move ι to the right. By CSIReadS, any read step that reads from a shard, must access a shard

whose local time is greater than the snapshot time of the transaction and all versions in the shard with times smaller than the snapshot time must be committed successfully. This means, the read step ι can be moved to the right, since any future steps will not affect the read. The full proof is in appendix A.7 on page 212. ■

Proposition 6.36 (Left mover: Clock-SI preparation and commit steps)**.** *Assume a Clock-SI trace $\phi \in AClockTrace$, two transitions with labels ι, ι', and a time-tick trace segment ϕ_1 such that*

$$\begin{aligned}\phi = \phi' \xrightarrow{\iota''} \left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}\right), \mathtt{I} \xrightarrow{\iota} \phi_2 \xrightarrow{\iota'} \phi'' \\ \wedge \mathtt{TickAndSnapshot}\left(\phi_1, r\right) \wedge (\iota' = (cl, r, (\mathtt{P}, k, v), n'', n) \vee \iota' = (cl, r, n, \mathtt{E})) \\ \wedge \exists cl', r', r'', l', k', v', n', n_1.\, cl \neq cl' \wedge r \neq r'' \wedge n_1 \geq n \\ \wedge (\iota = (cl', r', (l', k', v'), n', n_1) \vee \iota = (cl', r'', n', \mathtt{S}) \vee \iota = (cl', r', n', \mathtt{E}) \vee \iota = (r'', n'))\end{aligned}$$

where the rest free variables are universally quantified. The transition labelled by ι, together with the time-tick trace segment ϕ_1, can be moved to the left, that is, there exists a new time-tick trace segment ϕ_2, a new label ι_1 and a new equivalent trace such that

$$\mathtt{TickAndSnapshot}\left(\phi_2, r\right) \wedge \left(\phi' \xrightarrow{\iota''} \phi_2 \xrightarrow{\iota'} \_ \xrightarrow{\iota_1} \phi''\right) \simeq \phi. \tag{6.12}$$

Proof sketch. We know that ι must come from a client that differs from the client of ι'.

(1) Consider the cases that ι is a read and write step. Since any future steps will not affect any read and write step, thus, it is safe to move ι' and ϕ_1 to the left. In Eq. (6.12), $\iota_1 = \iota$.

(2) If ι is a transaction snapshot step, then ι must starts at a shard that is different from the shard of ι'. Therefore, it is safe to move ι' and ϕ_1 to the left. In Eq. (6.12), $\iota_1 = \iota$.

(3) If ι is a time-tick step, then ι must on a shard that is different from the shard of ι'. Therefore, it is safe to move ι' and ϕ_1 to the left. In Eq. (6.12), $\iota_1 = \iota$.

(4) Consider ι is a transaction preparation step. If this is a preparation step on a shard that is different a from from the shard of ι', it is safe to move ι' and ϕ_1 to the left. For this case, $\iota_1 = \iota$ in Eq. (6.12). Consider this is a preparation step on the same shard of ι'. By the hypothesis, the transaction of $\iota = (cl', r', (l, k', v'), n', n_1)$ must commit after the transaction of ι', thus it is safe to alter the preparation time in ι to a later time as the same as the preparation time in ι'. This means change ι to $\iota_1 = (cl', r', (l, k', v'), n, n_1)$ This allows us to move ι_1 and ϕ_1 to the left.

(5) Consider ι is a transaction commit step. This case is similar to the case of preparation step. In Eq. (6.12), $\iota_1 = \iota$.

The full proof is given in appendix A.7 on page 214. ■

6.2.5 Verification: Trace Refinement

By Theorem 6.34, it is enough to only consider annotated normalised traces. We verify Clock-SI protocol by first converting normalised Clock-SI traces to our kv-store program traces. We then show that these kv-store program traces can be obtained under the execution test for snapshot isolation $\mathtt{ET}_{\mathtt{PSI}}$.

Definition 6.37 (Conversion of Clock-SI traces to kv-store program traces)**.** *Given an annotated normalised Clock-SI trace ϕ, the kv-store program trace induced by the trace, written* $\mathsf{ClockToKVTrace}\,(\phi)$*, is defined in Fig. 6.15. The auxiliary predicate* $\mathtt{CLOCKAtomic}\,(\phi, cl, r, n)$*, and the auxiliary functions for extracting static program, fingerprint and view,* ClockStaticProg*,* ClockFp *and* ClockView*, are defined in Fig. 6.15.*

The base case for ClockToKVTrace convert the initial configuration of Clock-SI to the initial configuration in kv-store. The first inductive case, directly drop the next label ι, if ι is a snapshot ($\mathtt{S}$) or time tick step, because this step is transparent to client. The second inductive case convert a sequence of steps from a transaction in Clock-SI to an atomic step in our kv-store semantics. Assume a trace

$$\phi \xrightarrow{\iota} \phi' \xrightarrow{\iota'} \left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}'\right), \mathtt{I}$$

where:

(1) label ι is the first internal step of a transaction, that is, $\iota = (cl, r, (l, k, v), m, n) \wedge l \in \{\mathtt{R}, \mathtt{W}\}$, and m and n is the snapshot time and the commit time of this transaction;

(2) the trace segment $\phi_1 = \mathsf{Last}\,(\phi) \xrightarrow{\iota} \phi' \xrightarrow{\iota'} \left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}'\right), \mathtt{I}$ contains and only contains all the internal steps (with the same commit time n) of this transaction, yet including arbitrary steps of snapshot and time tick steps, captured by $\mathtt{CLOCKAtomic}\,(\phi_1, cl, r, n)$ predicate; and

(3) label ι' is the commit step of this transaction, also defined inside $\mathtt{CLOCKAtomic}$ predicate.

Recall that in an annotated normalised Clock-SI trace $\mathtt{CLOCKAtomic}\,(\phi_1, cl, r, n)$, We know that the read, write and preparation steps of a transaction can only be interfered by time-tick or snapshot steps. The predicate $\mathtt{CLOCKAtomic}\,(\phi', cl, r, n)$ states that the trace segment ϕ' contains all the read, write and preparation steps of the same transaction from the client cl with the same coordinator r and the commit time n.

We now explain the detail of the second inductive case. First, the transactional identifier $\tilde{t}_{cl}^{(n,m)}$ in the encoding kv-store trace, defined in Def. 6.38, is annotated with the client cl that commits the transaction, and the commit n and snapshot time m of the transaction. Note that it is useful to annotate the transaction identifiers with the snapshot times, however, it does not affect the encoding. Clock-SI identifiers in a store must be unique, because transactions for a client must have unique commit times. The detail is given in Prop. A.22 on page 216.

$$\mathsf{ClockToKVTrace}\left(\left(\mathcal{R}_0, \tilde{\mathcal{E}}_0, \mathcal{E}_0\right), \mathtt{P}_0\right) \stackrel{def}{=} \left((\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}_0), \mathtt{P}_0\right) \text{ where } \mathcal{U}_0 = \left(\lambda cl \in \mathsf{Dom}\left(\tilde{\mathcal{E}}_0\right).\, u_0\right)$$

$$\mathsf{ClockToKVTrace}\left(\phi \xhookrightarrow{\iota} \left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}'\right), \mathtt{I}\right) \stackrel{def}{=} \mathsf{ClockToKVTrace}\left(\phi\right) \texttt{ where } \iota = (cl, r, n, \mathtt{S}) \vee \iota = (r, n)$$

$$\mathsf{ClockToKVTrace}\left(\phi \xhookrightarrow{\iota} \phi' \xhookrightarrow{\iota'} \left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}'\right), \mathtt{I}\right) \stackrel{def}{=} \texttt{let } \eta = \mathsf{ClockToKVTrace}\left(\phi\right) \texttt{ and } \left(\left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}\right), \mathtt{P}\right) = \mathsf{Last}\left(\eta\right)$$
$$\texttt{in } \eta \xrightarrow[\mathtt{ET}]{(cl,u,\mathcal{F})} \left(\mathcal{K}', \mathcal{U}\left[cl \mapsto u'\right], \mathcal{E}'\right), \mathsf{ClockStaticProg}\left(\mathtt{I}\right)$$
$$\texttt{where } \iota = (cl, r, (l, k, v), m, n) \wedge l \in \{\mathtt{R}, \mathtt{W}\} \wedge \phi_1 = \mathsf{Last}\left(\phi\right) \xhookrightarrow{\iota} \phi' \xhookrightarrow{\iota'} \left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}'\right), \mathtt{I} \wedge \texttt{CLOCKAtomic}\left(\phi_1, cl, r, n\right)$$
$$\wedge\, \mathcal{F} = \mathsf{ClockFp}\left(\emptyset, \phi_1\right) \wedge u = \mathsf{ClockView}\left(\mathcal{K}, m\right) \wedge \mathcal{K}' = \mathsf{UpdateKV}\left(\mathcal{K}, u, \mathcal{F}, t_{cl}^{n,m}\right) \wedge u' = \mathsf{ClockView}\left(\mathcal{K}', n\right)$$

$$\texttt{CLOCKAtomic}\left(\phi, cl, r, n\right) \stackrel{def}{\Leftrightarrow} \exists m.\, m = |\phi| \wedge \forall \tilde{\alpha}_0, \cdots, \tilde{\alpha}_{m-1}, \iota_1, \cdots, \iota_{m-1}.\, \phi = \tilde{\alpha}_0 \xrightarrow{\iota_1} \tilde{\alpha}_1 \xrightarrow{\iota_2} \cdots \xrightarrow{\iota_{m-1}} \tilde{\alpha}_{m-1}$$
$$\Rightarrow \bigwedge_{0<i<m-1} \exists l \in \{\mathtt{R}, \mathtt{W}, \mathtt{P}\}.\, \exists cl', r', k, v, n'.\, (\iota_i = (cl, r, (l, k, v), n', n) \vee \iota_i = (cl', r, n', \mathtt{S}) \vee \iota_i = (r, n))$$
$$\wedge\, \iota_{m-1} = (cl, r, n, \mathtt{E})$$

$$\mathsf{ClockStaticProg}\left(\mathtt{I}\right) \stackrel{def}{=} \lambda cl \in \mathsf{Dom}\left(\mathtt{I}\right).\, \mathsf{ClockStaticCmd}\left(\mathtt{I}(cl)\right)$$

$$\mathsf{ClockStaticCmd}\left(\mathtt{R}\right) \stackrel{def}{=} \begin{cases} [\,\mathtt{T}\,] & \text{if } \mathtt{R} = [\,\mathtt{T}\,]^{\mathcal{F},\mathcal{B}}_{n,r}, \\ [\,\mathtt{T}\,]\,;\mathtt{C} & \text{if } \mathtt{R} = [\,\mathtt{T}\,]^{\mathcal{F},\mathcal{B}}_{n,r}\,;\mathtt{C}, \\ \mathtt{C} & \text{otherwise.} \end{cases}$$

$$\mathsf{ClockView}\left(\mathcal{K}, n\right) \stackrel{def}{=} \lambda k.\, \{i \mid \exists m, cl.\, \mathsf{WriterOf}\left(\mathcal{K}\left(k, i\right)\right) = t_{cl}^{m} \wedge m \leq n\}$$

$$\mathsf{ClockFp}\left(\mathcal{F}, \left(\mathcal{R}, \tilde{\mathcal{U}}, \tilde{\mathcal{E}}\right), \mathtt{P}\right) \stackrel{def}{=} \mathcal{F}$$

$$\mathsf{ClockFp}\left(\mathcal{F}, \phi \xhookrightarrow{\iota} \left(\mathcal{R}, \tilde{\mathcal{U}}, \tilde{\mathcal{E}}\right), \mathtt{I}\right) \stackrel{def}{=} \begin{cases} \mathsf{ClockFp}\left(\mathcal{F}, \phi\right) \lessdot\!\!\lhd \left(l, k, v\right) & \iota = (cl, r, (l, k, v), n, n') \wedge l \in \{\mathtt{R}, \mathtt{W}\} \\ \mathsf{ClockFp}\left(\mathcal{F}, \phi\right) & \text{otherwise} \end{cases}$$

Figure 6.15: Definitions of ClockToKVTrace, ClockStaticProg, ClockFp and ClockView functions, and definition of `CLOCKAtomic` predicate

Definition 6.38 (Clock-SI transaction identifiers)**.** *The set of* Clock-SI transaction identifiers, $\textsc{ClockTxID} \ni \tilde{t}$, *is defined by:*

$$\textsc{ClockTxID} \stackrel{def}{=} \left\{ \tilde{t}_{cl}^{(n,m)} \;\middle|\; cl \in \mathit{CID} \wedge n, m \in \textsc{ClockTime} \right\}.$$

Given two identifiers $\tilde{t}_{cl}^{(n,m)}, \tilde{t}_{cl'}^{(n',m')}$, *the order* $\tilde{t}_{cl}^{(n,m)} \sqsubseteq \tilde{t}_{cl'}^{(n',m')}$ *is defined by*

$$\tilde{t}_{cl}^{(n,m)} \sqsubseteq \tilde{t}_{cl'}^{(n',m')} \stackrel{def}{=} n \leq n' \vee (n = n' \wedge m \leq m').$$

Second, the views before, u, and after, u', the update are determined by the snapshot, n and commit time, m respectively, that is, $u = \mathsf{ClockView}\,(\mathcal{K}, m)$ and $u' = \mathsf{ClockView}\,(\mathcal{K}', n)$ in Fig. 6.15. The function $\mathsf{ClockView}\,(\mathcal{K}, m)$ defined in Fig. 6.15 collects all versions with timestamps smaller than m in the $\mathcal{K}$. The view u is well-formed, because transactions are committed at the same time to all shards. The detail is given in Prop. A.24 on page 217.

Third, the fingerprint of this transaction, $\mathcal{F}$ contains all the internal read and write operations of this transaction, via function $\mathcal{F} = \mathsf{ClockFp}\,(\emptyset, \phi_1)$ in Fig. 6.15. Given an initial fingerprint $\mathcal{F}_0 = \emptyset$, function $\mathsf{ClockFp}\,(\mathcal{F}_0, \phi)$ composites the read and write operations in ϕ' using the fingerprint combination operator $\lll$. This final fingerprint $\mathcal{F} = \mathsf{ClockFp}\,(\mathcal{F}_0, \phi)$ is well-formed, that is, it contains the first read of each key and the last write of each key. Detail is given in Prop. A.25 on page 217.

Last, given $\eta = \mathsf{ClockToKVTrace}\,(\phi)$ as the trace converted from ϕ as shown in Fig. 6.15, the second inductive case of $\mathsf{ClockToKVTrace}$, append a new transaction $(cl, u, \mathcal{F})$ to trace η updating the kv-store to $\mathcal{K}'$ with view $u = \mathsf{ClockView}\,(\mathcal{K}, m)$ and fingerprint $\mathsf{ClockFp}\,(\emptyset, \phi_1)$, and then updating the view of client cl to $u' = \mathsf{ClockView}\,(\mathcal{K}', n)$.

This kv-store trace defined by $\eta = \mathsf{ClockToKVTrace}\,(\phi)$, is correctly encoded ϕ, and satisfies the read atomic $\top$, by showing that: (1) every transition in the program kv-store $\eta = \mathsf{ClockToKVTrace}\,(\phi)$ is well-formed; and (2) if a version exists in $\mathsf{Last}\,(\eta)$, then it also exists in $\mathsf{Last}\,(\phi)$, and vice versa.

Theorem 6.39 (Well-formed Clock-SI centralised kv-store)**.** *Assume an annotate normalised Clock-SI trace* ϕ *and the kv-store trace induced by the Clock-SI trace* $\eta = \mathsf{ClockToKVTrace}\,(\phi)$. *Let* $(\mathcal{K}, \mathcal{U}, \mathcal{E}, \mathtt{P}) = \mathsf{Last}\,(\eta)$ *and* $(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}, \mathtt{I}) = \mathsf{Last}\,(\phi)$. *The final kv-store* $\mathcal{K}$ *is well-formed, and* $\mathcal{K}$ *only contains versions that exist in* $\mathcal{R}$, *and vice versa, that is,*

$$\begin{aligned}
&\forall k, i, v, cl, n, m, T.\, \mathcal{K}\,(k, i) = (v, \tilde{t}_{cl}^{(n,m)}, T) \Rightarrow \exists r, \tilde{\mathcal{K}}, i'. \\
&\quad r = \mathsf{ShardOf}_{\mathcal{R}}\,(k) \wedge (\tilde{\mathcal{K}}, _) = \mathcal{R}\,(r) \wedge \tilde{\mathcal{K}}\,(k, i') = (v, n, _), && (6.13) \\
&\forall k, r, \tilde{\mathcal{K}}, i, v, n.\, r = \mathsf{ShardOf}_{\mathcal{R}}\,(k) \wedge (\tilde{\mathcal{K}}, _) = \mathcal{R}\,(r) \wedge \tilde{\mathcal{K}}\,(k, i) = (v, n, _) \Rightarrow \\
&\qquad\qquad \exists i', cl, m, T.\, \mathcal{K}\,(k, i) = (v, \tilde{t}_{cl}^{(n,m)}, T). && (6.14)
\end{aligned}$$

Proof sketch. We prove this by induction on the trace ϕ. In the base case, Eqs. (6.13) and (6.14)

trivially hold. In the inductive case, by the definition of $\mathsf{ClockToKVTrace}$, we need to prove that the new kv-store, $\mathcal{K}' = \mathsf{UpdateKV}\left(\mathcal{K}, u, \mathcal{F}, \tilde{t}_{cl}^{(n,m)}\right)$, as the result of committing a new transaction $\tilde{t}_{cl}^{(n,m)}$ with fingerprint $\mathcal{F}$ and view cl, is still well-formed. Note that we have the following results:

(1) by inductive hypothesis, the kv-store $\mathcal{K}$ is well-formed;

(2) the pre-view u is well-formed on $\mathcal{K}$, the fingerprint $\mathcal{F}$ is well-formed fingerprint, and the transaction identifier $\tilde{t}_{cl}^{(n,m)}$ must be a unique, of which details are given in Props. A.22, A.24 and A.25; and

(3) client local times in Clock-SI monotonically increases (Lemma A.23), hence $\tilde{t}_{cl}^{(n,m)} \in \mathsf{NextTxID}\left(\mathcal{K}, cl\right)$.

Given the above and Theorem 4.21, the new kv-store $\mathcal{K}'$ is well-formed. Then, Eqs. (6.13) and (6.14) can be derived by the definition of $\mathsf{ClockToKVTrace}$, since the function does not create or remove any versions. The full proof is given in appendix A.7 on page 218. ■

By the result of Theorem 6.39, kv-store program traces induced by annotated normalised Clock-SI traces are valid traces that satisfy $\mathtt{ET}_{\top}$. Now we prove that each step in these kv-store traces satisfies $\mathtt{ET}_{\mathtt{SI}}$. The key is (1) to link the snapshot and commit times of all transactions, to relations WR, WW and RW; and to show that in the Clock-SI protocol, the snapshot and commit times of individual transaction lead to views that satisfy closure property of $\mathtt{ET}_{\mathtt{SI}}$ presented in Fig. 4.6a.

Theorem 6.40 (Clock-SI traces satisfying snapshot isolation)**.** *Given an annotated normalise Clock-SI trace ϕ, the kv-store trace induced by the Clock-SI trace $\eta = \mathsf{ClockToKVTrace}\left(\phi\right)$ is a trace that can be obtained with the execution test for* $\mathtt{SI}$.

Proof sketch. Consider a step in the trace η:

$$\left(\mathcal{K}, \mathcal{U}, \mathcal{E}\right), \mathtt{P} \xrightarrow{(cl,u,\mathcal{F})}_{\mathtt{ET}} \left(\mathcal{K}', \mathcal{U}\left[cl \mapsto u\right], \mathcal{E}'\right), \mathtt{P}'.$$

By the definition of $\mathtt{ET}_{\mathtt{SI}}$, we need to prove $\mathtt{ET}_{\mathtt{MR}}$, $\mathtt{ET}_{\mathtt{RYW}}$ and

$$\mathtt{PreClosed}\left(\mathcal{K}', u, \left(\left(\mathsf{WR}_{\mathcal{K}'} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}'}\right)^{+}; \mathsf{RW}_{\mathcal{K}'}^{-1}\right)^{*}\right) \tag{6.15}$$

$$\mathtt{PreClosed}\left(\mathcal{K}, u, \bigcup_{(\mathtt{W},k,\_)\in\mathcal{F}} \mathsf{WW}_{\mathcal{K}'}^{-1}\left(k\right)\right) \tag{6.16}$$

$\mathtt{ET}_{\mathtt{MR}}$ can be derived by Lemma A.23 that client local time monotonically increases, and the definition of $\mathsf{ClockView}$. $\mathtt{ET}_{\mathtt{RYW}}$ can be derived by CSICommit, where the client local time is set

to be the commit time, and then by the definition of ClockView: all the versions committed by the client must be included in the new view. Given the property of normalised traces:

(1) if two transactions $\left(\tilde{t}_{cl}^{(n,m)}, \tilde{t}_{cl'}^{(n',m')}\right) \in R$ for $R \in \{\mathsf{WR}, \mathsf{WW}, \mathsf{SO}\}$, we have the commit time of the first transaction n is smaller than the snapshot time of the second transaction m', thus $n < m' < n'$; and

(2) if two transactions $Tuple\tilde{t}_{cl}^{(n,m)}, \tilde{t}_{cl'}^{(n',m')} \in \mathsf{RW}$, we have the snapshot time of the first transaction m is smaller than the commit time of the second transaction n', that is $m < n'$.

Recall that the snapshot time is always smaller than commit time. If $\left(\tilde{t}_{cl}^{(n,m)}, \tilde{t}_{cl'}^{(n',m')}\right) \in \left((\mathsf{WR}_{\mathcal{K}'} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}'})^{+}; \mathsf{RW}_{\mathcal{K}'}^{-1}\right)^{*}$ then $n < n'$. Then by the definition of ClockView, we prove Eq. (6.15). Last, by the CSIPrepare, when a version is set to preparation stage, there is no conflict write, then Eq. (6.16) holds. The full detail is given in appendix A.7 on page 219. ■

We verify two protocols, COPS in Section 6.1 and Clock-SI in Section 6.2, via trace refinement in this chapter. Because system engineers did not have a formal definition of their protocols, but only pseudo-code, we first give formal operation semantics that captured these two protocols. The traces in these specific semantics is fine-grained in a sense that transactions are not executed in atomic steps. It is not direct to encode these fine-grained traces into our kv-store traces, hence we convert the traces to normalised traces where transactions are not interleaving by other transactions, and then convert normalised traces to kv-store traces, by squashing several transactional steps in normalised traces to atomic steps in kv-store traces. Last, we show that the kv-store traces satisfy our definitions of causal consistency and snapshot isolation respectively. The key steps of these proofs are to link the synchronisation mechanisms in COPS and Clock-SI, to the closure properties of causal consistency and snapshot isolation presented in Fig. 4.6a, respectively.

We have shown that our state-based operational semantics is useful in term of verifying implementation protocols:

(1) it is easier to map the system states in the specific protocols to our abstract kv-stores and views in comparison to the graph-based states, such as dependency graphs and abstract executions in Chapter 5, and [Nagar and Jagannathan, 2018]; and

(2) it is more piratical to map the semantics/pseudo-code of the specific protocols to steps in our kv-store operational semantics, in comparison with the declarative semantics in Chapter 5.

Chapter 7

Applications: Invariants of Client Programs

In Chapter 6, we show that our semantics can be used to verify implementation protocols. In this chapter, we focus on the client and use our operational semantics to prove the invariant properties of several client libraries. Clients only need to know the interface, our state-based abstract semantics, without knowing any details of the implementations. Hence, we show that our semantics is an ideal abstract interface to link implementations and clients.

Clients for specific applications often cannot commit arbitrary transactions, but via a set of pre-defined APIs, namely, *transactional client library*, L. These APIs are transactional operations which can be used by its clients to access the underlying kv-store. For instance, the counter library on a key k discussed in Chapter 3 is defined by:

$$\texttt{Counter}\,(k) \stackrel{def}{=} \{\texttt{Inc}\,(\texttt{x}, k)\,, \texttt{Read}\,(\texttt{x}, k) \mid \texttt{x} \in \textsc{Var}\}\,.$$

Note that $\texttt{Inc}\,(\texttt{x}, k)$ uses the $\texttt{x}$ to return the new value. A program $\texttt{P}$ is a *client program* of L, written $\texttt{P} \in L$, if and only if the transactional calls in $\texttt{P}$ are to operations of L.

In this chapter, we will show several libraries are robust against $\texttt{PSI}$ and $\texttt{SI}$ respectively. In general, robustness states that all the reachable states under a weak consistency model, are also reachable in serialisability. In our kv-store semantics, the robustness, defined in Def. 7.2, states that the set of reachable kv-stores of a library, defined in Def. 7.1, is a subset of $\mathsf{ConsisModel}\,(\texttt{SER})$.

Definition 7.1 (Reachable kv-stores of a library)**.** *The set of reachable kv-stores of a library L under execution test $\texttt{ET}$, written $[\![L]\!]_{\texttt{ET}}$, is defined by: $[\![L]\!]_{\texttt{ET}} \stackrel{def}{=} \bigcup_{\texttt{P}\in L} [\![\texttt{P}]\!]_{\texttt{ET}}$, where $[\![\_]\!]_{\texttt{ET}}$ is defined in Def. 4.24.*

Definition 7.2 (Robustness)**.** *A library L is* robust *with respect to an execution test $\texttt{ET}$, if and only if $[\![L]\!]_{\texttt{ET}} \subseteq \mathsf{ConsisModel}\,(\texttt{ET}_{\texttt{SER}})$, where $\mathsf{ConsisModel}$ is defined in Def. 4.27.*

The correctness of many client libraries can be derived from robustness, because many client programs are designed with respect to serialisability. Hence, it is an important property for many libraries. In contrast to previous work on checking robustness [Bernardi and Gotsman, 2016; Cerone and Gotsman, 2016; Cerone et al., 2017; Nagar and Jagannathan, 2018] by either examining the *full history* [Nagar and Jagannathan, 2018] or using static-analysis techniques that cannot be extended to support client sessions [Bernardi and Gotsman, 2016; Cerone and Gotsman, 2016; Cerone et al., 2017], we encode robustness as an invariant property and establish it at each step. This means that: (1) we do not need to check the entire execution history; and (2) we take client sessions into account.

In Section 7.1 we prove the robustness of the single-counter library discussed in Chapter 3 is against `PSI`. In Section 7.2.2 and Section 7.2.3, we prove the robustness of a multi-counter library and a banking library of Alomari et al. [2008] against `SI`. We do the latter by proving that multi-counter and bank libraries satisfy a general invariant property, which we call *`WSI`-safe*. In Section 7.2.1, we show that `WSI`-safe guarantees robustness against our new proposed model `WSI`. Because `WSI` is weaker than `SI`, hence any robustness proof against `WSI` directly implies robustness against stronger models such as `SI`.

We further use our operational semantics to prove *library-specific* properties. In Section 7.3, we show that a lock library is correct under `PSI`, in a sense that it satisfies the *mutual exclusion guarantee* under `PSI`, even though it is not robust against `PSI`. To do this, we simply encode such library-specific guarantees as invariants of the library, and establish them at each step, as described above. By contrast, establishing such library-specific properties using the existing techniques is more difficult. This is because unlike the kv-stores in our operational semantics, existing techniques do not directly record the library *state*, but they record full execution traces, making them less amenable for reasoning about such properties.

7.1 Robustness: A Single Counter Library against `PSI`

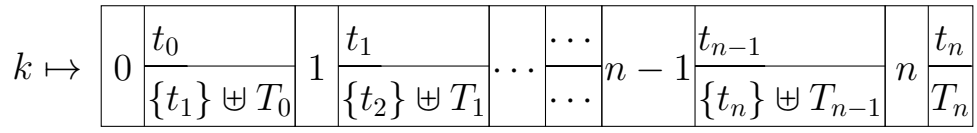
We first present Theorem 7.3, states a acyclic property on kv-stores $\mathcal{K} \in \mathsf{ConsisModel}\,(\mathtt{ET}_{\mathtt{SER}})$ that are reachable under serialisability. Specifically, it states that any reachable kv-store $\mathcal{K}$ under $\mathtt{ET}_{\mathtt{SER}}$ contains no cycle in the relation $(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{RW}_{\mathcal{K}})^{+}$. This result can be directly derived from the well-known result in [Adya, 1999].

Theorem 7.3 (Serialisable kv-stores)**.** *For all kv-stores $\mathcal{K}$,*

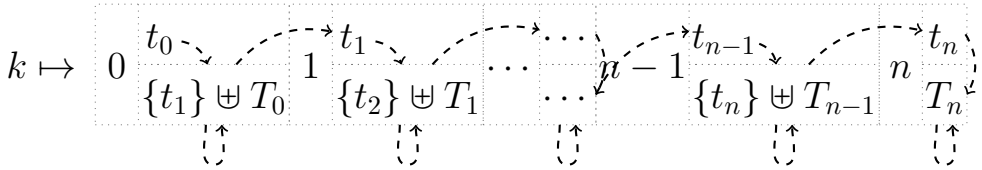
$$\forall t \in \mathit{TxID}.\, (t,t) \notin (\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{RW}_{\mathcal{K}})^{+} \Leftrightarrow \mathcal{K} \in \mathsf{ConsisModel}\,(\mathtt{ET}_{\mathtt{SER}})$$

Proof. Assume a kv-store $\mathcal{K}$ such that

$$\forall t \in \mathrm{TxID}.\, (t,t) \notin (\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{RW}_{\mathcal{K}})^{+}. \tag{7.1}$$



(a) Invariant of the reachable the single counter library under PSI



(b) The relation $\mathsf{WR} \cup \mathsf{SO} \cup \mathsf{WW} \cup \mathsf{RW}$ on the reachable kv-stores in Fig. 7.1a

Figure 7.1: Single counter library

By the definition of KToD in Def. 5.2 Eq. (7.1) holds, if and only if, $\mathcal{G} = \mathsf{KToD}(\mathcal{K})$, and

$$\forall t \in \mathrm{TxID}.\, (t,t) \notin (\mathsf{WR}_{\mathcal{G}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{G}} \cup \mathsf{RW}_{\mathcal{G}})^{+}. \tag{7.2}$$

By [Adya, 1999], Eq. (7.2) holds, if and only if, $\mathcal{G}$ is a reachable dependency graph under SER, and then by the definition of DToK in Def. 5.3, $\mathsf{DToK}(\mathcal{G}) \in \mathsf{ConsisModel}(\mathtt{ET}_{\mathtt{SER}})$. By the bijection between kv-stores and dependency graphs (Theorem 5.4), we then have $\mathcal{K} = \mathsf{DToK}(\mathsf{KToD}(\mathcal{K})) \in \mathsf{ConsisModel}(\mathtt{ET}_{\mathtt{SER}})$. ■

Using Theorem 7.3, we only need to check the state of the abstract kv-store so to determine if this kv-store can be obtained under serialisability, without re-constructing a kv-store trace under serialisability. More specifically, To prove the robustness of a library L, we prove that for every program $\mathtt{P} \in L$, every transition of every trace of P does not introduce cycle.

Given Theorem 7.3, it is easy to see that single-counter library is *not* robust against CC. Since the kv-store depicted in Fig. 3.1d is allowed under CC, it is not a serialisable kv-store due to the cycle $t' \xrightarrow{\mathsf{RW}} t \xrightarrow{\mathsf{WW}} t'$.

As PSI satisfies (UA), we know that if a transaction t updates k (by calling $\mathtt{Inc}(\mathtt{x}, k)$) and writes a new version ν to k, then it must have read the version of k immediately preceding ν: that is, $\forall t \in \mathcal{K}.\, \forall i \in \mathbb{N}.\, t = \mathsf{WriterOf}(\mathcal{K}(k,i)) \Rightarrow t \in \mathsf{ReadersOf}(\mathcal{K}(k,i-1))$. Moreover, as PSI enforces monotonic reads (MR), the order in which clients observe the versions of k by calling $\mathtt{Read}(\mathtt{x}, k)$ is consistent with the order of versions in $\mathcal{K}(k)$. As such, the kv-stores $\mathcal{K} \in [\![\mathtt{Counter}(k)]\!]_{\mathtt{PSI}}$ have the invariant depicted in Fig. 7.1a and defined below:

$$\mathcal{K}(k) = [(0, t_0, \{t_1\} \uplus T_0), (1, t_1, \{t_2\} \uplus T_1), \cdots, (n-1, t_{n-1}, \{t_n\} \uplus T_{n-1}), (n, t_n, T_n)]$$

where $t_i \notin T_j$ for any i, j.

Theorem 7.4 (Robustness of $\mathtt{Counter}(k)$ against PSI)**.** *Any reachable kv-store of the single-counter library under the execution test of* PSI*, that is,* $\mathcal{K} \in [\![\mathtt{Counter}(k)]\!]_{\mathtt{PSI}}$*, contains no cycle*

in the relation $(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{RW}_{\mathcal{K}})^+$*, that is:*

$$\forall t \in \mathcal{K}.\, (t, t) \notin (\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{RW}_{\mathcal{K}})^+.$$

Proof. Let prove this by extending $(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO})^+$ to a total order $\dashrightarrow$, depicted in Fig. 7.1b. First we pick a total order over all client identifiers, written $cl \xrightarrow{\mathtt{CT}} cl'$. We consider all the read-only transactions T_i for an index i. For any transactions $t, t' \in T_i$. If t, t' come from the same client, then $t \dashrightarrow_1 t'$ if $t \xrightarrow{\mathsf{SO}} t'$. Otherwise, they are from different clients, and then $t \dashrightarrow_1 t'$ if $t = t^{-}_{cl}$, $t' = t^{-}_{cl'}$ and $cl \xrightarrow{\mathtt{CT}} cl'$. Note that SO is acyclic, which means that the relation $\dashrightarrow_1$ is a total over T_i. Second, let consider the increment transactions. We define $t \dashrightarrow_2 t'$, if $t = t_i$, $t' = t_j$ and $i < j$. Third, for a version in an index i, we define $t \dashrightarrow_3 t'$, if $t = t_i$ and $t' \in T_i$; or $t \dashrightarrow_3 t'$, if $t \in T_i$ and $t' = t_{i+1}$. Last, the relation $\dashrightarrow$ is defined by: $\dashrightarrow = (\dashrightarrow_1 \cup \dashrightarrow_2 \cup \dashrightarrow_3)^+$, depicted in Fig. 7.1b. It is trivial that $(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{RW}_{\mathcal{K}})^+ \in \dashrightarrow$, which contains no cycle. This is because $t_i \notin T_j$ for all i, j. ■

7.2 Robustness for SI

Snapshot isolation SI is a well-known consistency to programmers, due to the good performance. Much work focuses on checking robustness against SI, due to the popularity of SI, such as [Bernardi and Gotsman, 2016; Cerone and Gotsman, 2016; Cerone et al., 2017; Nagar and Jagannathan, 2018]. However, those checks rely on the entire history of programs. We present a state-based approach in this section, by simply checking the *shape* of transactions from a library L, in a sense that, if transactions from a library L are either read-only transactions, or transactions that always read and write a set of keys simultaneously, then the library L guarantees robustness against WSI and hence SI. We call this shape property WSI-*safe* (Section 7.2.1). We then show that a multi-counter library and bank library satisfy this property in Section 7.2.2 and Section 7.2.3, respectively.

7.2.1 WSI Safe

Given a library L, we say the library is WSI-*safe*, if all the reachable kv-store satisfies Eqs. (7.3) to (7.5).

Definition 7.5 (WSI-safe)**.** *A library L is* WSI-safe*, if and only if, any reachable kv-store*

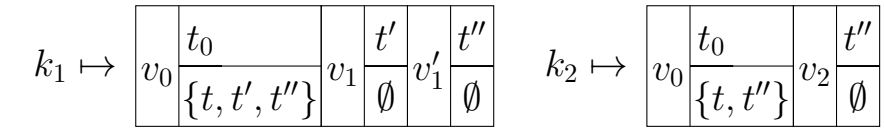


Figure 7.2: A kv-store under read atomic $\top$ that satisfies WSI-safe

$\mathcal{K} \in [\![L]\!]_{\mathtt{ET}_{\mathtt{WSI}}}$ *satisfies the following conditions:*

$$\exists \mathtt{P} \in \textsc{Progs}.\, \mathcal{K} \in [\![\mathtt{P}]\!]_{\mathtt{WSI}} \wedge \forall t \in \textsc{TxID}.\, \forall k \in \textsc{Key}.\, \forall i \in \mathbb{N}.$$

$$\begin{pmatrix} \forall i' \in \mathbb{N}.\, t \in \mathsf{ReadersOf}\left(\mathcal{K}\left(k, i\right)\right) \wedge t \neq \mathsf{WriterOf}\left(\mathcal{K}\left(k, i'\right)\right) \\ \quad \Rightarrow \forall k' \in \textsc{Key}.\, \forall i'' \in \mathbb{N}.\, t \neq \mathsf{WriterOf}\left(\mathcal{K}\left(k', i''\right)\right) \end{pmatrix} \tag{7.3}$$

$$\wedge \left(t \neq t_0 \wedge t = \mathsf{WriterOf}\left(\mathcal{K}\left(k, i\right)\right) \Rightarrow \exists i' \in \mathbb{N}.\, t \in \mathsf{ReadersOf}\left(\mathcal{K}\left(k, i'\right)\right)\right) \tag{7.4}$$

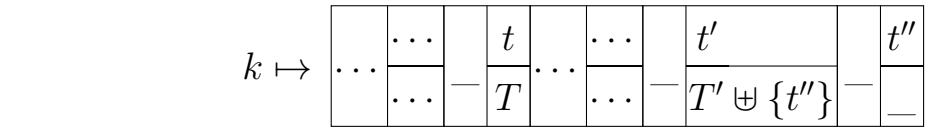
$$\wedge \begin{pmatrix} t \neq t_0 \wedge t = \mathsf{WriterOf}\left(\mathcal{K}\left(k, i\right)\right) \wedge \exists k' \in \textsc{Key}.\, \exists i' \in \mathbb{N}. \\ \quad t \in \mathsf{ReadersOf}\left(\mathcal{K}\left(k', i'\right)\right) \Rightarrow \exists i'' \in \mathbb{N}.\, t = \mathsf{WriterOf}\left(\mathcal{K}\left(k', i'\right)\right) \end{pmatrix} \tag{7.5}$$

The WSI-safe is a property on kv-store. Eq. (7.3) state that if a transaction, for example t in Fig. 7.2, reads from a key k without writing to k, then the transaction must be a read-only transaction. Eq. (7.4) state that if a transaction, for example t' in Fig. 7.2, writes to k, then it must also read from it, a property known as *no-blind-write*. Last, Eq. (7.5) states that if a transaction, for example t'' in Fig. 7.2, writes to k, then it must also write to all keys it reads. We call Eqs. (7.4) and (7.5) together *strict no-blind-writes*. Given a library L, when all its reachable kv-stores under WSI satisfy WSI-safe, then L must be robust against WSI, because, first, read-only transactions can be safely re-ordered, without affect other transactions. Second, the rest transactions must satisfy *strict no-blind-writes*, and they cannot concurrently write to the same key due to UA from PSI. For example, transactions t' and t'' concurrently write the key k_1 in Fig. 7.2, which is disallowed by UA. With these two intuition in mind, we now give the formal proof in Theorem 7.6. We prove this result by replacing certain relations given the WSI-safe property, and then showing a contradiction.

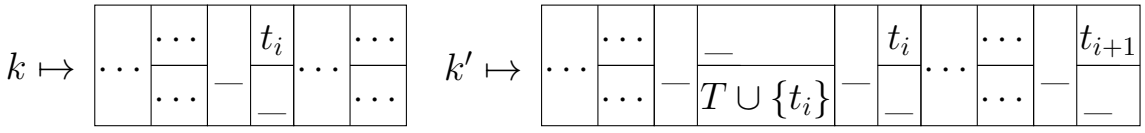
Theorem 7.6 (Robustness of WSI)**.** *Any* WSI*-safe library is robust against* WSI*.*

Proof. Assume a reachable kv-store $\mathcal{K} \in [\![L]\!]_{\mathtt{WSI}}$ for a library ι that is WSI-safe. Then by Def. 7.2 and Theorem 7.3 it is sufficient to prove acyclicity of the relation $(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{RW}_{\mathcal{K}})^+$. By Prop. 7.7 the relation $(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO})^+$ must be acyclic. Now consider the relation $(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}})^+$. Assume two transactions t, t' such that $(t, t') \in \mathsf{WW}_{\mathcal{K}}$. By Eq. (7.4), if transaction t writes a version, it must read the immediate previous version,

$$\forall k \in \textsc{Key}.\, \forall i \in \mathbb{N}.\, t = \mathsf{WriterOf}\left(\mathcal{K}\left(k, i+1\right)\right) \Rightarrow t \in \mathsf{ReadersOf}\left(\mathcal{K}\left(k, i\right)\right). \tag{7.6}$$



(a) $t \xrightarrow{\mathsf{WR}} \cdots \xrightarrow{\mathsf{WR}} t' \xrightarrow{\mathsf{WR}} t''$ replaces $t \xrightarrow{\mathsf{WW}} t''$



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

Figure 7.3: `WSI`-safety

By the definition of $\mathsf{WW}_{\mathcal{K}}$, there exists two versions, i^{th} and j^{th} versions of a key k,

$$t = \mathsf{WriterOf}\left(\mathcal{K}\left(k, i\right)\right) \wedge t' = \mathsf{WriterOf}\left(\mathcal{K}\left(k, j\right)\right) \wedge i < j.$$

Therefore, by Eq. (7.4), we have $(t, t') \in {\mathsf{WR}_{\mathcal{K}}}^{+}$ depicted in Fig. 7.3a. This means that the relation $(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}})^{+}$ is acyclic. We now prove acyclicity of the relation $(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{RW}_{\mathcal{K}})^{+}$ by contradiction. Assume a transaction t such that

$$t \xrightarrow{(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{RW}_{\mathcal{K}})^{+}} t$$

Since $(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}})^{+}$ cannot have cycle, it must be the case that the cycle contains $\mathsf{RW}_{\mathcal{K}}$. There exists t_1 to t_n such that

$$t = t_1 \wedge t = t_n \wedge t_1 \xrightarrow{R^*} t_2 \xrightarrow{\mathsf{RW}_{\mathcal{K}}} t_3 \xrightarrow{R^*} \cdots \xrightarrow{\mathsf{RW}_{\mathcal{K}}} t_{n-1} \xrightarrow{R^*} t_n \tag{7.7}$$

where $R = \mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}}$. We now convert some edges in the cycle; this yields equivalent cycle.

(1) $t_i \xrightarrow{\mathsf{RW}_{\mathcal{K}}} t_{i+1}$ **for t_i writing a key k.** By the definition of $\mathsf{RW}_{\mathcal{K}}$, transaction t_i must read a key k' that is overwritten by t_{i+1}, that is, $t_i \in \mathsf{ReadersOf}\left(\mathcal{K}\left(k', j\right)\right)$ for some index j. Since t_i wrote the key k, by Eq. (7.5), transaction t_i must also write the key k'. Then by Eq. (7.5) and $[\![\mathtt{P}]\!]_{\mathtt{UA}} \subseteq [\![\mathtt{P}]\!]_{\mathtt{WSI}}$, transaction t_i must also write the $(j+1)^{\text{th}}$ version of the key k': that is, $t_i \in \mathsf{WriterOf}\left(\mathcal{K}\left(k', j+1\right)\right)$, and therefore $t_i \xrightarrow{\mathsf{WW}_{\mathcal{K}}} t_{i+1}$. This result is illustrated in Fig. 7.3b.

After replace all the possible $\mathsf{RW}_{\mathcal{K}}$ edges to $\mathsf{WW}_{\mathcal{K}}$ in the cycle in Eq. (7.7), the rest $\mathsf{RW}_{\mathcal{K}}$ edges must start from a read only transaction: that is, for any transactions t', t'' in the new cycle,

$$t' \xrightarrow{\mathsf{RW}_{\mathcal{K}}} t'' \Rightarrow \forall k'' \in \textsc{Key}.\, \forall z \in \mathbb{N}.\, t' \neq \mathsf{WriterOf}\left(\mathcal{K}\left(k'', z\right)\right). \tag{7.8}$$

This means that any RW edge must start from a read-only transaction.

(2) $\cdots \xrightarrow{\mathsf{RW}} t_i \xrightarrow{R^*} t_{i+1} \xrightarrow{\mathsf{RW}} \cdots$ **in the new cycle.** Transaction t_i at least wrote a key, but by Eq. (7.8), the transaction t_{i+1} is a read-only transaction, thus $t_i \neq t_{i+1}$. This means that $\cdots \xrightarrow{\mathsf{RW}} t_i \xrightarrow{R^+} t_{i+1} \xrightarrow{\mathsf{RW}} \cdots$.

(3) $t_i \xrightarrow{\mathsf{WW}_{\mathcal{K}}} t_{i+1}$. By Eq. (7.4) and $[\![\mathtt{P}]\!]_{\mathtt{UA}} \subseteq [\![\mathtt{P}]\!]_{\mathtt{WSI}}$, we know that $t_i \xrightarrow{\mathsf{WR}_{\mathcal{K}}{}^+} t_{i+1}$ depicted in Fig. 7.3a.

After steps (1) to (3), we have cycle of the form:

$$t = t_1 = t'_1 \wedge t' = t_n = t'_m \wedge t'_1 \xrightarrow{R'^*} t'_2 \xrightarrow{\mathsf{RW}} t'_3 \xrightarrow{R'^+} \cdots \xrightarrow{\mathsf{RW}} t'_{m-1} \xrightarrow{R'^*} t'_m$$

for some transactions t'_1 to t'_m and $R' = \mathsf{WR} \cup \mathsf{SO}$. Note that $\left\{t'_j \;\middle|\; 1 \leq j \leq m\right\} \subseteq \{t_i \mid 1 \leq i \leq n\}$ for t_i defined in Eq. (7.7). This means that $t \xrightarrow{((\mathsf{WR} \cup \mathsf{SO}); \mathsf{RW}^?)^*} t$. Because of $[\![\mathtt{P}]\!]_{\mathtt{CP}} \subseteq [\![\mathtt{P}]\!]_{\mathtt{WSI}}$ and Prop. 7.8, these must be no cycle in the relation $((\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}); \mathsf{RW}^?_{\mathcal{K}})^+$, which contradicts with the assumption. Therefore, the relation $(\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}} \cup \mathsf{RW}_{\mathcal{K}})^+$ is acyclic. ■

In the proof for Theorem 7.6, we use two acyclicity properties on kv-stores. We prove these properties as invariants: (1) a kv-store $\mathcal{K}$ reachable under $\mathtt{ET}_\top$ contains no $(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}})$ cycles (Prop. 7.7), where $\mathtt{ET}_\top$ is the most permissive execution test defined in Fig. 4.6a; and (2) a kv-store $\mathcal{K}$ reachable under $\mathtt{ET}_{\mathtt{CP}}$ contains no $((\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}); \mathsf{RW}^{-1}_{\mathcal{K}})$ cycles (Prop. 7.8). The first result, Prop. 7.7, is a general result on our kv-store semantics stating that there is no out-of-order executions between transactions $(t, t') \in \mathsf{SO}$ or $(t, t') \in \mathsf{WR}$. The second result, Prop. 7.8, can be derived from the definition of $\mathtt{CP}$, which requires that view must be closed with respect to relation $((\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}); \mathsf{RW}^{-1}_{\mathcal{K}})$.

Proposition 7.7. *Any kv-store* $\mathcal{K} \in \mathsf{ConsisModel}\,(\mathtt{ET}_\top)$ *satisfies* $(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}})^+ \cap \mathtt{Id} = \emptyset$.

Proof sketch. From the definition of $\mathsf{ConsisModel}$ defined in Def. 4.27 we know a kv-store $\mathcal{K} \in \mathsf{ConsisModel}\,(\mathtt{ET}_\top)$ must be reachable with a given program. This means that Prop. 7.7 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: $(\mathcal{K}, \mathcal{U}, \mathcal{E})\,, \mathtt{P} \xrightarrow{(cl,u,\mathcal{F})}_{\mathtt{ET}} (\mathcal{K}', \mathcal{U}', \mathcal{E}')\,, \mathtt{P}'$, where $\mathcal{K}$ contains no $R \stackrel{def}{=} (\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}})$ 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 $(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}'})^+ \cap \mathtt{Id} = \emptyset$. ■

Proposition 7.8. *Any kv-store* $\mathcal{K} \in \mathsf{ConsisModel}\,(\mathtt{ET}_{\mathtt{CP}})$ *satisfies* $((\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}); \mathsf{RW}_{\mathcal{K}}^{?})^{+} \cap \mathtt{Id} = \emptyset$.

Proof sketch. We proceed as in the proof of Prop. 7.7. For the inductive case, consider

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

where $\mathcal{K}$ contains no $R \stackrel{def}{=} ((\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}); \mathsf{RW}_{\mathcal{K}}^{?})$ cycles by the inductive hypothesis. Recall that $R_1; R_2 \stackrel{def}{=} \{(a, c) \mid \exists b.\, (a, b) \in R_1 \wedge (b, c) \in R_2\}$. 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) [**Case**: $t \xrightarrow{R} t_1 \xrightarrow{R} \cdots \xrightarrow{R} t_n \xrightarrow{R} t$] This cycle cannot exist as t is the last transaction in $\mathcal{K}'$. More concretely, as in Prop. 7.7 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 $t' \in \mathcal{K}$.

(2) [**Case**: $t_1 \xrightarrow{R} \cdots \xrightarrow{R} t_n \xrightarrow{(\mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}'})} t \xrightarrow{\mathsf{RW}_{\mathcal{K}'}} t_1$] From $\mathtt{ET}_{\mathtt{CP}}$ 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. ■

7.2.2 Robustness: A Multiple Counters Library against WSI

A multi-counter library on a set of keys K is defined by $\mathtt{Counters}\,(K) \stackrel{def}{=} \bigcup_{k \in K} \mathtt{Counter}\,(k)$. Let take $K = \{k_1, k_2\}$ as an example. In this multi-counter library, a client is allowed to access both k_1 and key_2 in a session, however, not in a transaction, for example

$$\begin{aligned} cl_1 : \mathtt{Inc}\,(\mathtt{x}, k_1)\,; \mathtt{Inc}\,(\mathtt{x}, k_2)\,; \mathtt{Read}\,(\mathtt{x}, k_1)\,; \mathtt{Read}\,(\mathtt{x}, k_1) \parallel \\ cl_2 : \mathtt{Inc}\,(\mathtt{y}, k_1)\,; \mathtt{Inc}\,(\mathtt{y}, k_2)\,; \mathtt{Read}\,(\mathtt{y}, k_1)\,; \mathtt{Read}\,(\mathtt{y}, k_2)\,. \end{aligned}$$

One possible result of the program above is shown in Fig. 7.4. Client cl_1 executes its two increments on k_1 and k_2 first, then cl_2 increments these two keys afterwards. Note that, under WSI and hence PSI, client cl_2 includes all versions of k_1 and k_2 respectively, when incrementing (reading and then writing) these two keys. Last, both clients execute their last reads on both keys respectively, where clients, especially cl_1, are allowed to read old values as long as the view satisfies the closure on relation $\mathsf{SO}; \mathsf{RW}^{?} \cup \mathsf{WR}; \mathsf{RW}^{?} \cup \mathsf{WW}$, as required by WSI defined in Fig. 4.6a. Hence, cl_1 read the second versions of both keys, instead of the latest versions.

As discussed in Chapter 3, the multi-counter library is not robust against PSI. We show that this library is WSI-safe in Theorem 7.9, and therefore robust against WSI and all stronger models such as SI. It is easy to see that: any transaction yielding from $\mathtt{Inc}\,(\mathtt{x}, k)$, must read and then write the key k, hence satisfies Eqs. (7.4) and (7.5), and Eq. (7.3) is irrelevant; and any

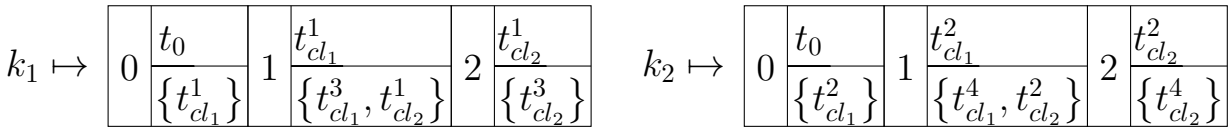


Figure 7.4: An example kv-store of multi-counter library under `WSI`

transaction yielding from $\texttt{Read}(\texttt{x}, k)$ must be a read-only transaction on key k, hence satisfies Eq. (7.3), and Eqs. (7.4) and (7.5) are irrelevant.

Theorem 7.9 (Robustness of multi-counter against `WSI`)**.** *For a set of keys K, the multi-counter library* $\texttt{Counters}(K)$ *is robust against* `WSI`*.*

Proof. It is sufficient to show that $\texttt{Counters}(K)$ is `WSI`-safe. Let $(\mathcal{K}_0, \mathcal{U}_0)$ be an initial configuration such that $\mathsf{Dom}(\texttt{P}_0) \subseteq \mathsf{Dom}(\mathcal{U}_0)$, and let $\mathcal{E}$ be a client environment such that $\mathsf{Dom}(\texttt{P}_0) \subseteq \mathsf{Dom}(\mathcal{E})$. Let η be a trace obtained by execution a program $\texttt{P} \in \texttt{Counters}(K)$ with the initial state being $(\mathcal{K}_0, \mathcal{U}_0), \mathcal{E}$. We prove that the final kv-store $\mathcal{K}$ in the trace η satisfies Eqs. (7.3) to (7.5) by induction on the length of traces η.

(1) [**Case**: $\eta = (\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}), \texttt{P}_0$] The initial kv-store $\mathcal{K}_0$ trivially satisfies Eqs. (7.3) to (7.5).

(2) [**Case**: $\eta = \eta' \xrightarrow{\iota}_{\texttt{ET}} (\mathcal{K}, \mathcal{U}, \mathcal{E}), \texttt{P}$] If the next step ι is a local computation, then by the inductive hypothesis, $\mathcal{K}$ must be `WSI`-safe. Let $(\mathcal{K}', \mathcal{U}', \mathcal{E}'), \texttt{P}' = \mathsf{Last}(\eta')$ be the last configuration. Consider that ι is a transaction step.

(i) [**Case**: ι **from** $\texttt{Read}(\texttt{x}, k)$] The label ι must be $\iota = (cl, u, \mathcal{F})$ for some cl, view u and fingerprint $\mathcal{F}$ such that $\mathcal{F} = \{(\texttt{R}, k, v)\}$. The resulting kv-store is given by the following:

$$\mathcal{K} = \mathcal{K}'[k \mapsto \mathcal{K}'(k)[i \mapsto (v, t', T \uplus \{t\})]]$$

for $i = \mathsf{Max}(u(k))$ and $\mathcal{K}'(k, i) = (v, t', T)$. By the inductive hypothesis, in the new kv-store $\mathcal{K}$, transactions that already exist in $\mathcal{K}'$ satisfy Eqs. (7.3) to (7.5). Since the new transaction t is a read-only transaction, t satisfies Eq. (7.3), which state that if a transaction only read a key, then it disallowed to write to any key, and Eqs. (7.4) and (7.5) are irrelevant, of which the hypothesis contains conditions about writing to a key.

(ii) [**Case**: ι **from** $\texttt{Inc}(\texttt{x}, k)$] The label ι must be $\iota = (cl, u, \mathcal{F})$ for some cl, view u and fingerprint $\mathcal{F}$ such that $\mathcal{F} = \{(\texttt{R}, k, v), (\texttt{W}, k, v+1)\}$. Since transaction t writes to key k. Then by `UA` the view for the key must included all versions associated with the key, we have $u(k) = \{n \mid 0 \leq n < |\mathcal{K}'(k)|\}$. Then the resulting kv-store $\mathcal{K}$ is:

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

where $\mathcal{V} = (\mathcal{K}'(k)\,[i \mapsto (v, t', T \uplus \{t\})])$, $i = |\mathcal{K}'(k)| - 1$ and $\mathcal{K}'(k, i) = (v, t', T)$. As the new transaction t reads the latest version of k and writes a new version to k and the inductive hypothesis, the new kv-store $\mathcal{K}$ satisfies Eqs. (7.3) to (7.5). First, the hypothesis of Eq. (7.3) contains conditions about only reading a key, hence it is irrelevant here. Eq. (7.4) requires if a transaction write a key, k here, it must also read it, which is true here, and Eq. (7.5) talks about reading and writing to several keys in one transaction, which is irrelevant here. ■

7.2.3 Robustness: A Banking Library Against `WSI`

Alomari et al. [2008] present a banking library that they shown to be robust against `SI`. Alomari et al. [2008] informally argued this banking library is robust: (1) they identified a notion of dangerous dependency between transactions, which, they argued, can lead to violation of robustness of `SI`; and (2) they argued this banking example contains no such dangerous dependencies. We show that this library is also robust against `WSI` and hence `SI` by proving invariant property. The banking example is based on relational databases and has three tables: account, saving and checking.

Account	
Name	CID
Alice	00001
Bob	00002
...	...

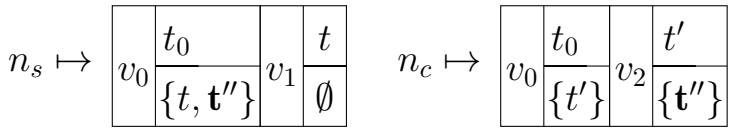
Saving	
CID	Balance
00001	33
00002	87
...	...

Checking	
CID	Balance
00001	-1
00002	5
...	...

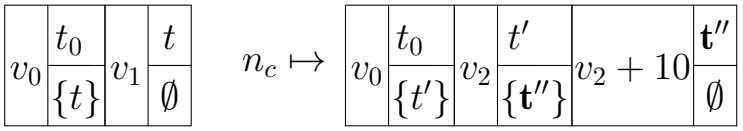
The account table maps customer names to customer IDs (`Account`(`Name`, `CID`)), the saving table maps customer IDs to their saving balances (`Saving`(`CID`, `Balance`)) and the checking table maps customer IDs to their checking balances (`Checking`(`CID`, `Balance`)). The balance of a saving account must be non-negative but a checking account may have a negative balance.

For simplicity, we encode the saving and checking tables as a kv-store, and forgo the account table as it is an immutable lookup table. We model each customer ID as an integer $n \in \mathbb{N}$ and assume that balances are integer values. We then define the key associated with customer n in the checking table as $n_c \stackrel{def}{=} 2n$; and define the key associated with n in the saving table as $n_s \stackrel{def}{=} 2n + 1$.

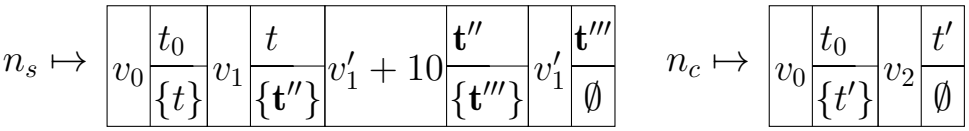
The banking library provides five transactional operations for accessing the database, where



(a) Client n reads the balance in transaction t''



(b) Client n deposits 10 pounds to its checking account in transaction identifier t'' and then withdraws 10 pounds in t'''



(c) Client n deposits 10 pounds to its saving account, $\texttt{DepositChecking}\,(n, \texttt{v})$, via transaction identifier t'', where $\texttt{v} = 10$

Figure 7.5: An example of the banking library, part 1

the `ret` is special variable that carries the return value or a transaction.

$$\texttt{Balance}\,(n) \stackrel{def}{=} [\ \texttt{x} := \texttt{rd}\,(n_s)\,;\ \texttt{y} := \texttt{rd}\,(n_c)\,;\ \texttt{ret} := \texttt{x} + \texttt{y}\ ]$$

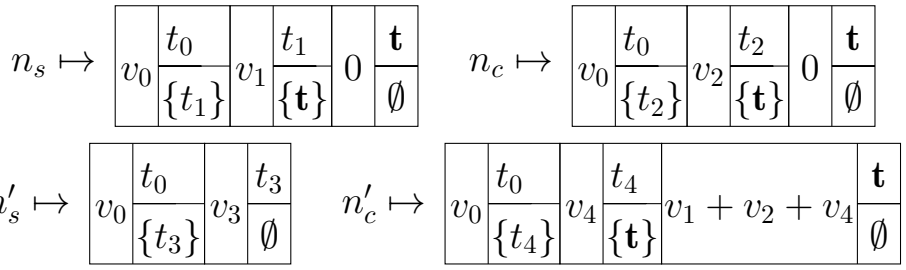
$$\texttt{DepositChecking}\,(n, \texttt{v}) \stackrel{def}{=} [\ \texttt{if}\,(\texttt{v} \geq 0)\ \{\ \texttt{x} := \texttt{rd}\,(n_c)\,;\ \texttt{wt}\,(n_c, \texttt{x} + \texttt{v})\ \}\ ]$$

$$\texttt{TransactSaving}\,(n, \texttt{v}) \stackrel{def}{=} [\ \texttt{x} := \texttt{rd}\,(n_s)\,;\ \texttt{if}\,(\texttt{v} + \texttt{x} \geq 0)\ \{\ \texttt{wt}\,(n_s, \texttt{x} + \texttt{v})\ \}\ ]$$

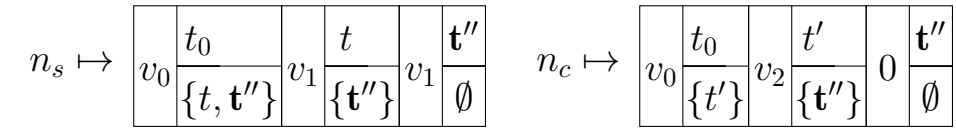
$$\texttt{Amalgamate}\,(n, n') \stackrel{def}{=} \begin{bmatrix} \texttt{x} := \texttt{rd}\,(n_s)\,;\ \texttt{y} := \texttt{rd}\,(n_c)\,;\ \texttt{z} := \texttt{rd}\,(n_c')\,; \\ \texttt{wt}\,(n_s, 0)\,;\ \texttt{wt}\,(n_c, 0)\,;\ \texttt{wt}\,(n_c', \texttt{x} + \texttt{y} + \texttt{z}) \end{bmatrix}$$

$$\texttt{WriteCheck}\,(n, \texttt{v}) \stackrel{def}{=} \begin{bmatrix} \texttt{x} := \texttt{rd}\,(n_s)\,;\ \texttt{y} := \texttt{rd}\,(n_c)\,; \\ \texttt{if}\,(\texttt{x} + \texttt{y} < \texttt{v})\ \{\ \texttt{wt}\,(n_c, \texttt{y} - \texttt{v} - 1)\ \} \\ \texttt{else}\ \{\ \texttt{wt}\,(n_c, \texttt{y} - \texttt{v})\ \} \\ \texttt{wt}\,(n_s, \texttt{x}) \end{bmatrix}$$

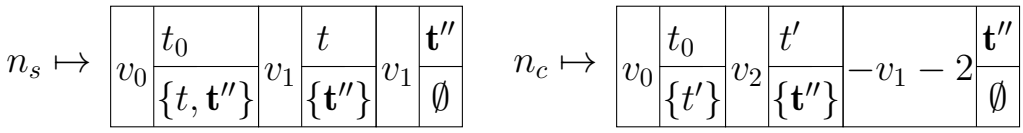
The $\texttt{Balance}\,(n)$ operation returns the total balance of customer n in `ret`. This is a read-only transaction on keys n_s and n_c, hence it is allowed to read old values under `WSI`, for example t'' in Fig. 7.5a. The $\texttt{DepositChecking}\,(n, \texttt{v})$ operation deposits `v` to the checking account of customer n when `v` is non-negative, by reading the checking account n_c and writing back a new value, for example t'' in Fig. 7.5c. Otherwise the checking account remains unchanged. When $\texttt{v} \geq 0$, operation $\texttt{TransactSaving}\,(n, \texttt{v})$ deposits `v` to the saving account of n, for example t'' in Fig. 7.5b. When $\texttt{v} < 0$, operation $\texttt{TransactSaving}\,(n, \texttt{v})$ withdraws `v` from the saving account of n only if the resulting balance is non-negative, for example t''' in Fig. 7.5b, otherwise the saving account remains unchanged. Note that both $\texttt{DepositChecking}\,(n, \texttt{v})$ and $\texttt{TransactSaving}\,(n, \texttt{v})$ must read the latest value of the checking account n_c and saving account n_s respectively under `WSI`, because the view must include all versions of n_c and n_s



(a) Client n amalgamates to client n' in transaction t



(b) Client n writes a check in transaction t'' with value v_2



(c) Client n write a check in transaction t'' with value $v_1 + v_2 + 1$

Figure 7.6: An example of the banking library, part 2

respectively, if a transaction wants to write it. The $\texttt{Amalgamate}\,(n, n')$ operation moves the combined checking and saving funds of consumer n to the checking account of customer n'. In this operation, a transaction will read and write checking and saving account of n, and checking account of n', for example t in Fig. 7.6a. Lastly, $\texttt{WriteCheck}\,(n, \mathtt{v})$ cashes a cheque of customer n in the amount $\mathtt{v}$ by deducting $\mathtt{v}$ from its checking account, for example t'' in Fig. 7.6b. However, if n does not hold sufficient funds, that is, the combined checking and saving balance is less than $\mathtt{v}$, customer n is penalised by deducting one additional pound, for example t'' in Fig. 7.6c. Alomari et al. [2008] argue that to make the banking library robust against SI, the $\texttt{WriteCheck}\,(n, \mathtt{v})$ operation must be strengthened by writing back the balance to the saving account (via $\mathtt{wt}\,(n_s, \mathtt{x})$), even though the saving balance is unchanged, as shown in Figs. 7.6b and 7.6c where transaction t'' read val_1 and write back the same value to n_s. This also matches our WSI-safety property: if t'' does not write back some value to n_s, this transaction will violate Eq. (7.5) in Def. 7.5. The banking library for a set of customers N, written $\texttt{Bank}\,(N)$, is defined by

$$\texttt{Bank}\,(N) \stackrel{def}{=} \left\{ \begin{array}{l} \texttt{Balance}\,(n)\,, \texttt{DepositChecking}\,(n, \mathtt{v})\,, \\ \texttt{TransactSaving}\,(n, \mathtt{v})\,, \texttt{Amalgamate}\,(n, n')\,, \\ \qquad\qquad\qquad\qquad \texttt{WriteCheck}\,(n, \mathtt{v}) \end{array} \middle| \; n, n' \in N \wedge \mathtt{v} \in \textsc{Var} \right\}$$

The banking library is more complex than the multi-counter library discussed in Section 7.2.2. Nevertheless, with the strengthen of $\mathtt{wt}\,(n_s, \mathtt{x})$ in $\texttt{WriteCheck}\,(n, \mathtt{v})$, all banking transactions are either read-only or satisfy the strictly-no-blind-writes property; the banking library is WSI-

safe. As such, we can prove its robustness against WSI in a similar fashion to that of the multi-counter library.

Theorem 7.10 (Robustness of the banking application against WSI)**.** *The bank library* $\texttt{Bank}\,(N)$ *is robust against* WSI*.*

Proof. It is sufficient to prove that any kv-store $\mathcal{K}$ of library $\texttt{Bank}\,(N)$ is WSI-safe. As $\texttt{Balance}\,(n)$ is read-only, it immediately satisfies Eq. (7.3) and Eqs. (7.4) and (7.5) are irrelevant. When $\texttt{v} \geq 0$, then the operation $\texttt{DepositChecking}\,(n, \texttt{v})$ both reads and writes n_c, and thus preserves Eqs. (7.4) and (7.5), and Eq. (7.3) is irrelevant. Similar, when $\texttt{v} \geq 0$, then $\texttt{DepositChecking}\,(n, \texttt{v})$ leaves the kv-store unchanged Eqs. (7.4) and (7.5) are trivially preserved, and Eq. (7.3) is irrelevant. Lastly, the $\texttt{TransactSaving}\,(n, \texttt{v})$, $\texttt{Amalgamate}\,(n, n')$ and $\texttt{WriteCheck}\,(n, \texttt{v})$ operations always read and write the keys they access, thus satisfying Eqs. (7.4) and (7.5), and Eq. (7.3) is irrelevant. ■

In our approach to proving robustness, we prove the invariant property in each step. For example, in the banking example, we only need to check the new transaction satisfies the Eqs. (7.3) to (7.5). In contrast, the previous approach based on execution graphs (dependency graphs) requires checking the shape of entire graphs.

7.3 Correctness: A Lock Pattern against PSI

A client program might still have the desired behaviour even if it is not robust. We now show a lock library is correct under UA. The distributed lock library providing $\texttt{lock}\,(k)$, $\texttt{tryLock}\,(k)$ and $\texttt{unlock}\,(k)$ operations on the key k, is defined by:

$$\begin{aligned}
\texttt{tryLock}\,(k) &\stackrel{def}{=} [\ \texttt{x} := \texttt{rd}\,(k)\,;\ \texttt{if}\,(\texttt{x} = 0)\ \{\texttt{wt}\,(k, \texttt{CID}_{cl})\,;\ \texttt{m} := \texttt{true}\ \}\texttt{else}\{\ \texttt{m} := \texttt{false}\ \}\ ] \\
\texttt{lock}\,(k) &\stackrel{def}{=} \texttt{do}\{\ \texttt{m} := \texttt{false};\ \texttt{tryLock}\,(k)\ \}\texttt{until}(\texttt{m} = \texttt{true}) \\
\texttt{unlock}\,(k) &\stackrel{def}{=} [\ \texttt{wt}\,(k, 0)\,;\ m := \texttt{false}\ ]
\end{aligned}$$

The tryLock operation reads the value of the key k. If the value is zero (the lock is available), the operation sets it to the client ID, $\texttt{CID}_{cl}$, and assign true to the local variable m. Otherwise the transaction does not change k and assigns false to m. Note that m carries the boolean outside the transaction. The lock operation calls tryLock until it successfully acquires the lock. The unlock operation simply sets the key k to zero.

Consider the set of programs $\texttt{P}_{\texttt{LK}}$ where clients cl and cl' compete for the lock k:

$$\texttt{P}_{\texttt{LK}} \stackrel{def}{=} (cl : (\texttt{lock}\,(k)\,;\ ...;\ \texttt{unlock}\,(k))^* \parallel cl' : (\texttt{lock}\,(k)\,;\ ...;\ \texttt{unlock}\,(k))^*)$$

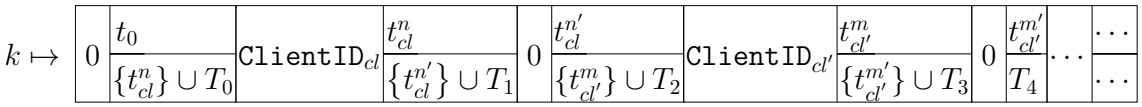


Figure 7.7: Correctness of lock patterns where $n < n'$ and $m < m'$

The locking library 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 and any stronger model such as SI. However, $\mathtt{P_{LK}}$ is not robust under UA and PSI, because the lock operation might read any old value of key k until it reads the up-to-date value of k and acquires the lock. Nevertheless, we show that $\mathtt{P_{LK}}$ is still correct under UA. We capture the correctness, that is, mutual exclusion: for all i, if $i > 0$, then:

$$\mathsf{ValueOf}\left(\mathcal{K}\left(k,i\right)\right) \neq 0 \Leftrightarrow \mathsf{ValueOf}\left(\mathcal{K}\left(k,i-1\right)\right) = 0 \tag{7.9}$$

$$\mathsf{ValueOf}\left(\mathcal{K}\left(k,i\right)\right) = \mathtt{CID}_{cl} \Rightarrow \exists n.\, \mathsf{WriterOf}\left(\mathcal{K}\left(k,i\right)\right) = \mathsf{WriterOf}\left(\mathcal{K}\left(k,i+1\right)\right) = t_{cl}^n. \tag{7.10}$$

It is straightforward to show that under UA, only one client can hold the key Eq. (7.9), and the same client releases the lock Eq. (7.10). This mutual exclusion property is illustrated in Fig. 7.7.

Theorem 7.11 (Mutual exclusion of the lock library under UA)**.** *Given the lock program,* $\mathtt{P_{Lock}}$*, defined by: for a set of clients CL, a key* k *and a critical section containing command* C*,*

$$\mathtt{P_{Lock}} \stackrel{def}{=} \lambda cl \in CL.\, (\mathtt{lock}\,(k)\,;\ \mathtt{C};\ \mathtt{unlock}\,(k))^*$$

where $k \notin \mathsf{fv}\,(\mathtt{C})$*, all reachable kv-stores* $\mathcal{K} \in [\![\mathtt{P_{Lock}}]\!]_{\mathtt{UA}}$ *satisfy Eqs.* (7.9) *and* (7.10)*.*

Proof. By the definition of $[\![\mathtt{P_{Lock}}]\!]_{\mathtt{UA}}$ defined in Def. 4.24, if $\mathcal{K} \in [\![\mathtt{P_{Lock}}]\!]_{\mathtt{UA}}$, then there exists a trace η with the final state $(\mathcal{K},\mathcal{U},\mathcal{E}),\mathtt{P}$: $\eta \in \textsc{PTraces}\,(\mathtt{P_{Lock}},\mathtt{UA}) \wedge (\mathcal{K},\mathcal{U},\mathcal{E}),\mathtt{P} = \mathsf{Last}\,(\eta)$. Let $\mathsf{FirstTr}\,(\mathtt{C})$ be the first tryLock or unlock transaction of the command C. We prove that that $\mathcal{K}$ satisfies Eqs. (7.9) and (7.10), and the following result:

$$\begin{aligned}\Big(\mathsf{ValueOf}\left(\mathcal{K}\left(k,|\mathcal{K}\left(k\right)|-1\right)\right) = 0 \Rightarrow \forall cl \in \mathsf{Dom}\,(\mathtt{P})\,.\,\mathsf{FirstTr}\,(\mathtt{P}\,(cl)) = \mathtt{tryLock}\,(k)\Big)\\ \wedge \left(\begin{array}{l}\exists cl, n.\, \mathcal{K}\left(k,|\mathcal{K}\left(k\right)|-1\right) = (\mathtt{CID}_{cl}, t_{cl}^n, \_) \Rightarrow \forall cl' \in \mathsf{Dom}\,(\mathtt{P})\,.\\ \quad (cl = cl' \Rightarrow \mathsf{FirstTr}\,(\mathtt{P}\,(cl)) = \mathtt{unlock}\,(k))\\ \quad \vee\, (cl \neq cl' \Rightarrow \mathsf{FirstTr}\,(\mathtt{P}\,(cl)) = \mathtt{tryLock}\,(k))\end{array}\right)\end{aligned} \tag{7.11}$$

by induction on the trace η. Eq. (7.11) means that: (1) if the latest value of k is zero (the first conjunction), then all clients are competing the lock; and (2) otherwise the latest value is $\mathtt{CID}_{cl}$ for a client cl (the second conjunction), then cl holds the lock and is responsible for releasing

the lock, and other clients are competing the lock.

(1) [**Base Case**: $\eta = (\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}_0), \mathtt{P}$] It is trivial that the configuration satisfies Eqs. (7.9) to (7.11).

(2) [**Inductive Case**: $\eta = \eta' \xrightarrow{\iota}_{\mathtt{ET}} (\mathcal{K}'', \mathcal{U}'', \mathcal{E}''), \mathtt{P}''$] Let $(\mathcal{K}', \mathcal{U}', \mathcal{E}'), \mathtt{P}' = \mathsf{Last}(\eta')$. By the inductive hypothesis, $\mathcal{K}', \mathtt{P}'$ satisfies Eqs. (7.9) to (7.11). If ι is a local computation $\iota = (cl, \bullet)$, we have $\mathcal{K}'' = \mathcal{K}'$ and $\mathcal{U}'' = \mathcal{U}'$. Therefore, the final configuration trivially satisfies Eqs. (7.9) to (7.11). Otherwise, ι is a transaction step. Assume $\iota = (cl, u, \mathcal{F})$, for a scheduled client cl with view u and fingerprint $\mathcal{F}$. This transaction might be a transaction of `tryLock`, `unlock`, or any transaction [`T`] in the critical section `C`. If the transaction is [`T`], because $k \notin \mathsf{fv}(\mathtt{C})$, we have $\mathcal{K}'(k) = \mathcal{K}''(k)$ and $\mathsf{FirstTr}(\mathtt{P}'(cl)) = \mathsf{FirstTr}(\mathtt{P}''(cl))$. This means $\mathcal{K}''$ and $\mathtt{P}''$ satisfy Eqs. (7.9) to (7.11). We now consider ι be a transaction step of `tryLock` or `unlock`. Given the value of the last version of k in $\mathcal{K}'$, we fix $\iota = (cl, u, \mathcal{F})$ and consider the following two possible cases.

(i) [**Case**: $\mathcal{K}'(k, |\mathcal{K}'(k)| - 1) = (0, t^{n'}_{cl'}, T')$] By Eq. (7.11), we have

$$\forall cl'' \in \mathsf{Dom}(\mathtt{P}') \,.\, \mathsf{FirstTr}(\mathtt{P}'(cl'')) = \mathtt{tryLock}(k)$$

and the next transaction for cl is $\mathtt{tryLock}(k)$ transaction.

(a) [**Case**: $u(k) = \{i \mid 0 \leq i < |\mathcal{K}'(k)| - 1\}$] The value for k in the snapshot $\sigma = \mathsf{Snapshot}(\mathcal{K}', u)$ must be zero: that is, $\sigma(k) = 0$. By the definition of `tryLock`, the final fingerprint will be $\mathcal{F}' = \{(\mathtt{R}, k, 0), (\mathtt{W}, k, \mathtt{CID}_{cl})\}$ where $\mathtt{CID}_{cl}$ is the unique client ID of cl. By the execution test for `UA`, this fingerprint $\mathcal{F}$ is allowed to commit with the given view u'. Let $\mathcal{K}'' = \mathsf{UpdateKV}(\mathcal{K}', u, \mathcal{F}, t)$ for a fresh $t^n_{cl} \in \mathsf{NextTxID}(\mathcal{K}', cl)$. We have $\mathcal{K}''(k, |\mathcal{K}(k)| - 1) = (\mathtt{CID}_{cl}, t^n_{cl}, \emptyset)$. By the definition of `tryLock`, we also have $\mathcal{E}''(cl)(\mathtt{m}) = \mathtt{true}$, which means $\mathsf{FirstTr}(\mathtt{P}''(cl)) = \mathtt{unlock}$. For other client $cl' \neq cl$, by the inductive hypothesis, we have $\mathsf{FirstTr}(\mathtt{P}''(cl')) = \mathtt{tryLock}$. Therefore, $\mathcal{K}''$ and $\mathtt{P}''$ satisfy Eqs. (7.9) to (7.11).

(b) [**Case**: $u(k) \subset \{i \mid 0 \leq i < |\mathcal{K}'(k)| - 1\}$] Note that the view u cannot be a view that leads to a snapshot $\sigma = \mathsf{Snapshot}(\mathcal{K}', u)$ in which the value of key k is zero: that is, $\sigma(k) = 0$. Because in this case, the fingerprint will be $\mathcal{F} = \{(\mathtt{R}, k, 0), (\mathtt{W}, k, \mathtt{CID}_{cl})\}$. However, it is disallowed to commit under `UA`. Because this fingerprint writes to key k, but the view for k does not contain all version of k. Thus, u must be a view such that $\mathsf{Snapshot}(\mathcal{K}', u)(k) \neq 0$. The final fingerprint will be $\{(\mathtt{R}, k, \mathsf{Snapshot}(\mathcal{K}', u)(k))\}$. Because it is a read-only transaction, it is trivial that this fingerprint is allowed to commit under `UA` and $\mathcal{K}'(k, |k| - 1) = \mathcal{K}''(k, |k| - 1)$. By the definition of `tryLock`, we also have $\mathcal{E}''(cl)(\mathtt{m}) = \mathtt{false}$, which means $\mathsf{FirstTr}(\mathtt{P}''(cl)) = \mathtt{tryLock}$. By the inductive hypothesis, $\mathcal{K}''$ and $\mathtt{P}''$ satisfy Eqs. (7.9) to (7.11).

(ii) [**Case**: $\mathcal{K}'(k, |\mathcal{K}'(k)| - 1) = (\mathtt{CID}_{cl'}, t^{n'}_{cl'}, T')$] By Eq. (7.11), we have two possible cases, depending on if $cl = cl'$.

(a) [**Case**: $cl = cl'$] By Eq. (7.11), the next transaction for cl is `unlock`. The final fingerprint $\mathcal{F}$ must be $\mathcal{F} = \{(\mathtt{W}, k, 0)\}$. In this case, the view must be $u\,(k) = \{i \mid |0 \leq i < |\mathcal{K}'(k)| - 1\}$, otherwise it is disallowed to commit by `UA`. Let $\mathcal{K}'' = \mathsf{UpdateKV}\,(\mathcal{K}', u, \mathcal{F}, t)$ for a fresh $t^n_{cl} \in \mathsf{NextTxID}\,(\mathcal{K}', cl)$. We have $\mathcal{K}''(k, |\mathcal{K}\,(k)| - 1) = (0, t^n_{cl}, \emptyset)$. By the definition of `unlock`, we have $\mathsf{FirstTr}\,(\mathtt{P}''(cl)) = \mathtt{tryLock}$. For other client $cl' \neq cl$, by the inductive hypothesis, we have $\mathsf{FirstTr}\,(\mathtt{P}''(cl')) = \mathtt{tryLock}$. Therefore, $\mathcal{K}''$ and $\mathtt{P}''$ satisfy Eqs. (7.9) to (7.11).

(b) [**Case**: $cl \neq cl'$] The next transaction for cl is $\mathtt{tryLock}\,(k)$ transaction, and the view u cannot be a view such that $\mathsf{Snapshot}\,(\mathcal{K}', u)\,(k) = 0$. Otherwise fingerprint will be $\mathcal{F} = \{(\mathtt{R}, k, 0)\,, (\mathtt{W}, k, \mathtt{CID}_{cl'})\}$, which disallowed to commit under `UA`. Thus, we have $\mathsf{Snapshot}\,(\mathcal{K}', u)\,(k) \neq 0$. By the definition of `tryLock`, the fingerprint $\mathcal{F} = \{(\mathtt{R}, k, \mathsf{Snapshot}\,(\mathcal{K}', u)\,(k))\}$. It is trivial that this fingerprint is allowed to commit under `UA`, because the transaction is a read-only transaction. By the definition of `tryLock`, we also have $\mathcal{E}''(cl)(\mathtt{m}) = \mathtt{false}$, which means $\mathsf{FirstTr}\,(\mathtt{P}''(cl)) = \mathtt{tryLock}$. This means that cl does not successfully lock the lock, and the lock still holds by cl'. Therefore, $\mathcal{K}''$ and $\mathtt{P}''$ satisfy Eqs. (7.9) to (7.11). ■

Theorem 7.11 guarantees the critical section protected by the lock *execute* as an non-interleaving block. For example, let us consider the following program:

$$\begin{aligned}\mathtt{P}_{\mathtt{example}} \stackrel{def}{=} (cl : (\mathtt{lock}\,(k)\,;\ t_1 : [\ \mathtt{T}_1\ ]\,; t_2 : [\ \mathtt{T}_2\ ]\,;\ \mathtt{unlock}\,(k)) \parallel \\ cl' : (\mathtt{lock}\,(k)\,;\ t_3 : [\ \mathtt{T}_3\ ]\,; t_4 : [\ \mathtt{T}_4\ ]\ \mathtt{unlock}\,(k)))\end{aligned}$$

where for brevity, we label the static code with dynamic transaction identifiers. The possible execution orders among these four transactions are either t_1, t_2, t_3, t_4 or t_3, t_4, t_1, t_2, but situation like t_1, t_3, t_2, t_4 cannot happen, because when cl executes t_1, the client cl must already hold the lock, and by Theorem 7.11, cl_2 cannot acquire the lock simultaneously, which is formally capture by the predicate $\mathtt{CSec}\,(\eta)$ defined in Theorem 7.12. The base case of $\mathtt{CSec}\,(\eta)$ simply states that the initial state satisfies `CSec` predicate. The first inductive case states that if a client cl does a local computation, or does not successfully acquire the lock, this step preserves `CSec` predicate. The second inductive case states that if a trace segment η' is protected by the lock, ι, and unlock ι' steps of client cl, then η', captured by $\mathtt{NoInterfere}\,(\eta', cl)$, cannot contains and lock and unlock steps from other clients cl'. Note that clients cl' are still allowed to read the value of lock k but will not be able to acquire the lock,

However, the lock library under `UA` does *not* guarantees, for example, t_4 observes the effect of t_1. This falls back to the main discussion of this thesis that transactions are allowed some level of weak behaviours, that is, one transaction t does not necessarily observe the effect of another

transaction t', although t happens before t'. In the case of UA, if a transaction does not contain any write operation, UA allows it to read old value without any constraint. This contradicts with programmer's intuition of a lock that is captured by Coherence in Theorem 7.12. The base case of Coherence states that the initial configuration satisfies the predicate. The first inductive case states that any steps that are not protected between the lock and unlock steps preserves the property of Coherence. The second inductive case, $\mathsf{Coherence}\left(\eta \xrightarrow{\iota}_{\mathtt{ET}} \eta' \xrightarrow{\iota'}_{\mathtt{ET}} (\mathcal{K}, \mathcal{U}, \mathcal{E}), \mathtt{P}\right)$ states that, if ι and ι' lock and unlock steps from client cl, then all the steps from cl in η' must observe all previous transactions in η, which is captured by $\mathsf{TCoherence}(\eta', cl, \{t \mid t \in \mathcal{K}'\})$ where $\mathcal{K}'$ is the last kv-store in η. The predicate $\mathsf{TCoherence}(\eta', cl, T)$ state that any transactions from cl, that updates $\mathcal{K}$ to $\mathcal{K}'$, must has a view u containing all the versions written by $\{t \mid t \in \mathcal{K}'\}$, that is, $\mathsf{GetView}(\mathcal{K}, T) \sqsubseteq u$. To handle this problem, this lock library must be executed in a stronger model than UA, any model that requires the view to be closed under SO, for example PSI:

(1) UA guarantees the atomicity of the critical section as shown in Theorem 7.11, which means, for example, a possible trace for $\mathtt{P}_{\mathtt{example}}$ is

$$\Gamma_0 \xrightarrow{t^{\mathtt{L}}_{cl_1}:(cl_1,u_1,\mathcal{F}_{\mathtt{lock}})}_{\mathtt{ET}} \Gamma_1 \xrightarrow{t_1:(cl_1,u_2,\mathcal{F}_1)}_{\mathtt{ET}} \Gamma_2 \xrightarrow{t_2:(cl_1,u_3,\mathcal{F}_2)}_{\mathtt{ET}} \Gamma_3 \xrightarrow{t^{\mathtt{U}}_{cl_1}:(cl_1,u_4,\mathcal{F}_{\mathtt{unlock}})}_{\mathtt{ET}}$$
$$\Gamma_4 \xrightarrow{t^{\mathtt{L}}_{cl_2}:\left(cl_2,u'_1,\mathcal{F}_{\mathtt{lock}}\right)}_{\mathtt{ET}} \Gamma_5 \xrightarrow{t_3:\left(cl_2,u'_2,\mathcal{F}_3\right)}_{\mathtt{ET}} \Gamma_6 \xrightarrow{t_4:\left(cl_2,u'_3,\mathcal{F}_4\right)}_{\mathtt{ET}} \Gamma_7 \xrightarrow{t^{\mathtt{U}}_{cl_2}:\left(cl_2,u'_4,\mathcal{F}_{\mathtt{unlock}}\right)}_{\mathtt{ET}} \Gamma_8$$

where for brevity, we label the step with transaction identifiers, and L and U stand for lock and unlock transactions.

(2) The closure under SO, that is, $\mathtt{PreClosed}(\mathsf{SO})$ as required by PSI, guarantees a critical section observes the effects of previous critical sections, because, for example, in the trace above, we know

$$t^{\mathtt{L}}_{cl_1} \xrightarrow{\mathsf{SO}} t_1 \xrightarrow{\mathsf{SO}} t_2 \xrightarrow{\mathsf{SO}} t^{\mathtt{U}}_{cl_1}$$

and (i) by Theorem 7.11, when the cl_2 acquires the lock, it must include the effect of $t^{\mathtt{U}}_{cl_1}$ in the view, (ii) by the closure of SO, the view must include the effect of t_1 and t_2 in its view; and (iii) by MR, the view of cl_2 must always include the effect of t_1 and t_2. Therefore, transactions t_3 and $txid_4$ will observe t_1 and t_2.

hence we prove this lock library is correct with respect to PSI, although it is not robust.

Theorem 7.12 (Value coherence of of the lock library under PSI)**.** *Recall the lock library in Theorem 7.11,*

$$\mathtt{P}_{\mathtt{Lock}} \stackrel{def}{=} \lambda cl \in CL.\, (\mathtt{lock}(k)\,;\ \mathtt{C};\ \mathtt{unlock}(k))^*$$

where $k \notin \mathsf{fv}(\mathtt{C})$. *Given a trace of* $\mathtt{P}_{\mathtt{Lock}}$ *under* PSI, *it must satisfy* $\mathtt{CSec}(\eta)$, *defined by:*

$$\mathtt{CSec}\left((\mathcal{K}_0,\mathcal{U}_0,\mathcal{E}_0),\mathtt{P}_{\mathtt{Lock}}\right) \overset{def}{\Leftrightarrow} \mathtt{true}$$

$$\mathtt{CSec}\left(\eta \xrightarrow{\iota}_{\mathtt{ET}} (\mathcal{K},\mathcal{U},\mathcal{E}),\mathtt{P}\right) \overset{def}{\Leftrightarrow} \mathtt{CSec}(\eta) \wedge \exists cl, u, \mathcal{F}. \\ (\iota = (cl,u,\mathcal{F}) \Rightarrow \mathcal{F} \neq \{(\mathtt{R},k,0),(\mathtt{W},k,\mathtt{CID}_{cl})\})$$

$$\mathtt{CSec}\left(\eta \xrightarrow{\iota}_{\mathtt{ET}} \eta' \xrightarrow{\iota'}_{\mathtt{ET}} (\mathcal{K},\mathcal{U},\mathcal{E}),\mathtt{P}\right) \overset{def}{\Leftrightarrow} \mathtt{CSec}(\eta) \wedge \exists cl,u,u'.\, \iota = (cl,u,\{(\mathtt{R},k,0),(\mathtt{W},k,\mathtt{CID}_{cl})\}) \\ \wedge \iota' = (cl,u',\{(\mathtt{W},k,0)\}) \wedge \mathtt{NoInterfere}(\eta',cl)$$

$$\mathtt{NoInterfere}\left((\mathcal{K},\mathcal{U},\mathcal{E}),\mathtt{P},cl\right) \overset{def}{\Leftrightarrow} \mathtt{true}$$

$$\mathtt{NoInterfere}\left(\eta \xrightarrow{\iota}_{\mathtt{ET}} (\mathcal{K},\mathcal{U},\mathcal{E}),\mathtt{P},cl\right) \overset{def}{\Leftrightarrow} \exists cl',u,\mathcal{F}.\, \iota = (cl',u,\mathcal{F}) \wedge cl \neq cl' \\ \Rightarrow \mathcal{F} \neq \{(\mathtt{R},k,0),(\mathtt{W},k,\mathtt{CID}_{cl'})\} \wedge \mathcal{F} \neq \{(\mathtt{W},k,0)\}$$

and must satisfy $\mathtt{Coherence}(\eta)$, *defined by:*

$$\mathtt{Coherence}\left((\mathcal{K}_0,\mathcal{U}_0,\mathcal{E}_0),\mathtt{P}_{\mathtt{Lock}}\right) \overset{def}{\Leftrightarrow} \mathtt{true}$$

$$\mathtt{Coherence}\left(\eta \xrightarrow{\iota}_{\mathtt{ET}} (\mathcal{K},\mathcal{U},\mathcal{E}),\mathtt{P}\right) \overset{def}{\Leftrightarrow} \mathtt{Coherence}(\eta) \wedge \exists cl,u,\mathcal{F}. \\ (\iota = (cl,u,\mathcal{F}) \Rightarrow \mathcal{F} \neq \{(\mathtt{R},k,0),(\mathtt{W},k,\mathtt{CID}_{cl})\})$$

$$\mathtt{Coherence}\left(\eta \xrightarrow{\iota}_{\mathtt{ET}} \eta' \xrightarrow{\iota'}_{\mathtt{ET}} (\mathcal{K},\mathcal{U},\mathcal{E}),\mathtt{P}\right) \overset{def}{\Leftrightarrow} \mathtt{Coherence}(\eta) \wedge \exists \mathcal{K}', cl,u,u'. \\ \iota = (cl,u,\{(\mathtt{R},k,0),(\mathtt{W},k,\mathtt{CID}_{cl})\}) \\ \wedge \iota' = (cl,u',\{(\mathtt{W},k,0)\}) \\ \wedge (\mathcal{K}',\_,\_,\_) = \mathsf{Last}(\eta) \\ \wedge \mathsf{TCoherence}(\eta',cl,\{t \mid t \in \mathcal{K}'\})$$

$$\mathsf{TCoherence}\left(((\mathcal{K},\mathcal{U},\mathcal{E}),\mathtt{P}),cl,T\right) \overset{def}{\Leftrightarrow} \mathtt{true}$$

$$\mathsf{TCoherence}\left(\left(\eta \xrightarrow{\iota}_{\mathtt{ET}} (\mathcal{K},\mathcal{U},\mathcal{E}),\mathtt{P}\right),cl,T\right) \overset{def}{\Leftrightarrow} \exists \mathcal{K}', u, \mathcal{F}.\, (\mathcal{K},\_,\_,\_) = \mathsf{Last}(\eta) \\ \wedge \iota = (cl,u,\mathcal{F}) \Rightarrow \mathsf{GetView}(\mathcal{K},T) \sqsubseteq u$$

Proof sketch. The first property, CSec, can be directly derived from Theorem 7.11. We prove the second property, Coherence, by induction on the trace.

(1) [**Base Case**: $(\mathcal{K}_0,\mathcal{U}_0,\mathcal{E}_0),\mathtt{P}_{\mathtt{Lock}}$] It trivially holds in this case.

(2) [**Inductive Case**: $\eta \xrightarrow{\iota}_{\mathtt{ET}} (\mathcal{K},\mathcal{U},\mathcal{E}),\mathtt{P}$ **with** $\iota = (cl,u,\mathcal{F}) \Rightarrow \mathcal{F} \neq \{(\mathtt{R},k,0),(\mathtt{W},k,\mathtt{CID}_{cl})\}$] We directly apply inductive hypothesis.

(3) **Inductive Case:** [$\eta \xrightarrow{\iota}_{\mathtt{ET}} \eta' \xrightarrow{\iota'}_{\mathtt{ET}} (\mathcal{K},\mathcal{U},\mathcal{E}),\mathtt{P}$ with $\iota = (cl,u,\{(\mathtt{R},k,0),(\mathtt{W},k,\mathtt{CID}_{cl})\})$ and $\iota' = (cl,u',\{(\mathtt{W},k,0)\})$] Let $\mathcal{K}'$ be the final kv-store of η, and $T = \{t \mid t \in \mathcal{K}'\}$ be the set of all transactions in $\mathcal{K}'$. Let $t^{\mathtt{L}}_{cl}$ be the transaction identifier corresponding to the lock step ι. We immediately know that, by UA,

$$\{t \mid \exists i.\, t = \mathsf{WriterOf}(\mathcal{K}'(k,i))\} \subseteq \mathsf{VisTrans}(u,\mathcal{K}')$$

and then by Theorem 7.11 and CSec,

$$\forall t \in T.\, \exists t' \in \{t'' \mid \exists i.\, t'' = \mathsf{WriterOf}\,(\mathcal{K}'(k,i))\}\,.\, t \xrightarrow{\mathsf{SO}} t'$$

which means any transaction t committed in $\mathcal{K}'$ must be before a transaction t' that write a version, specifically value zero, to the lock k. Last, by the closure on SO required by $R_{\mathtt{PSI}}$,

$$T \subseteq ({R_{\mathtt{PSI}}}^*)^{-1}\,(\mathsf{VisTrans}\,(u, \mathcal{K}'))\,,$$

hence $\mathsf{GetView}\,(\mathcal{K}, T) \sqsubseteq u$. We now prove TCoherence by induction on η'.

(i) [**Base Case**: $(\mathcal{K}'', \mathcal{U}'', \mathcal{E}'')\,, \mathtt{P}''$] It trivially holds for this case.

(ii) [**Base Case**: $\eta'' \xrightarrow{\iota''}_{\mathtt{ET}} (\mathcal{K}'', \mathcal{U}'', \mathcal{E}'')\,, \mathtt{P}''$ **with** $\iota = (cl, u'', \mathcal{F}'')$] Because of MR PSI, we know $\mathsf{GetView}\,(\mathcal{K}, T) \sqsubseteq u \sqsubseteq u''$. ■

In this chapter, we use our operation semantics to prove the invariant properties of client libraries:

(1) robustness of single-counter library under PSI, and multi-counter and bank libraries under SI;

(2) mutual exclusion and value coherence of lock library under PSI.

In comparison with client reasoning in declarative semantics, which works on the entire history of programs, for example [Nagar and Jagannathan, 2018; Cerone and Gotsman, 2016; Cerone et al., 2015a], our approach proves invariant properties on the states of kv-stores. More importantly, client reasoning techniques based on declarative semantics, dependency graphs and abstract executions as presented in Chapter 5, often focuses on the shape of the graphs, but not the states, while it is easier to use our approach to prove specific invariant on the states, for example the mutual exclusion of a lock library Section 7.3.

However, in this chapter, we only show small examples, which can be verified by hand-writing proofs. In the future, we aim to explore other invariant properties such as transaction chopping, and develop a tool, for example, a model checking tool, which allows up to reason larger examples such as TPC-C [TPCC, 1992] and RUBiS [RUBIS, 2008].

Chapter 8

Conclusion and Future Work

We have introduced a simple interleaving semantics for atomic transactions, based on a global, centralised kv-stores and partial client views in Chapter 4. Our operational semantics is parametrised by execution tests, which give rise to many consistency models, summarised in Fig. 4.6a. Our kv-stores track partial, yet the most important, histories of a database, which contain enough information to express all the well-known consistency models, for example, snapshot isolation, parallel snapshot isolation and causal consistency. Our kv-stores contain similar information as dependency graphs, as shown in Chapter 5. In contrast to declarative semantics where consistency models are defined by constraining the shape of the graphs, hence ruling out invalid graphs in Chapter 5, our novel client partial views on the kv-stores provide the ability to express consistency in an operational way. our execution tests constrain individual transitions in that only good transitions are allowed, more specifically, the views must satisfy `CanCommit` and `ViewShift` predicates, discussed in Section 4.2.4:

(1) `CanCommit` states that the client partial view, when the client commits a new transaction, must be closed with respect to a relation specified by the chosen consistency model; and

(2) after the update, `ViewShift` states that the view must be updated to a new view that must include certain versions.

Using our execution tests, we are able to capture the anomalous behaviour of many weak consistency models used in distributed key-value stores, however, bearing in mind the constraints that our transactions satisfy snapshot property and the last-write-wins policy, which are hard-wired in that: (1) a client partial view must include all or none of the versions written by another transaction, defined in the well-formedness of views Def. 4.6; (2) the snapshot induced by a view, only contains the latest version in the view of each key, defined in Def. 4.8; and (3) a transaction only commits its first read operation and last write operation per key to the kv-store, defined in the CAtomicTrans rule in Fig. 4.2.

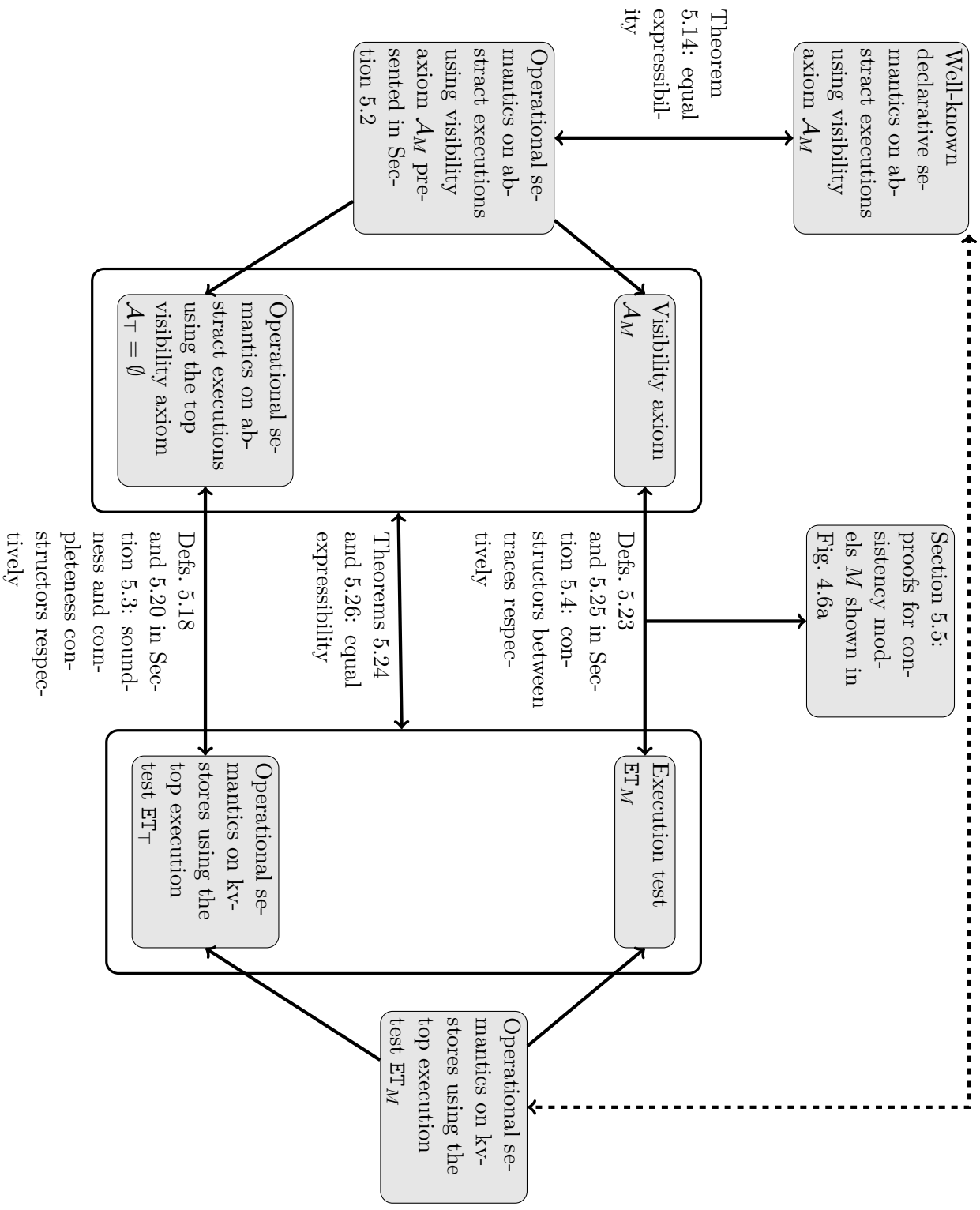


Figure 8.1: Summary of correctness proofs for definitions of consistency models M using execution tests in Fig. 4.6a

In Chapter 5, we have shown that our definitions using execution tests ET are equivalent to the well-known declarative definitions, using the visibility axioms $\mathcal{A}$ on abstract executions, where the latter is the de facto standard. This is a series of highly non-trivial proofs, summarised in

Fig. 8.1, because the declarative definitions rule out invalid states on the entire history of a program, but our operational definitions build up valid states step by step. The key step is the novel soundness and completeness constructors, which lift the equivalent conditions between ET and $\mathcal{A}$ to the equivalence between traces of kv-stores and abstract executions, and then, the equivalence between reachable kv-stores and abstract executions.

Many semantics [Adya, 1999; Burckhardt et al., 2012; Cerone et al., 2015a; Kaki et al., 2017; Crooks et al., 2017; Doherty et al., 2019; Nagar and Jagannathan, 2018; Koskinen et al., 2010; Koskinen and Parkinson, 2015] focus on one of the two challenges: verifying implementation protocols and reasoning about client programs. Our operational semantics, we believe, is the first general semantics that is suitable for both challenges. In Chapter 6, We have verified two protocols, COPS [Lloyd et al., 2011], a full replicated database that satisfies causal consistency, and Clock-SI [Du et al., 2013], a full partitioned database that satisfies snapshot isolation. We have verified these two protocols via trace refinement:

(1) we first provide a specific, formal semantics of these two protocols, because the authors [Lloyd et al., 2011; Du et al., 2013] only provided pseudo-codes;

(2) we refine the fine-grained traces of these specific, formal semantics to equivalent normalised traces, where transactions are not interleaved by other transactions;

(3) we encode the traces to our kv-store traces; and

(4) we then show that each step in the kv-store traces satisfies the desired consistency models.

Although every step is a non-trivial proof, but it is very standard trace refinement. More importantly, even though, we believe, abstract executions and the trace-based semantics proposed by Crooks et al. [2017] are suitable for verifying implementation, we did not find any works that show how to use them to verify implementation protocols.

In Chapter 7, we show how to use our semantics to reason about client programs. In contrast to [Nagar and Jagannathan, 2018; Cerone and Gotsman, 2016; Cerone et al., 2015a] that focus on analysing the shape of the final states in abstract executions or dependency graphs, We prove the invariant properties of client programs: (1) the robustness of a single-counter library against PSI; (2) the robustness of a multi-counter library and a banking library against SI via proving general conditions which guarantee the robustness of WSI, a new model that is slightly weaker than SI; and (3) the mutual exclusion of a lock library against PSI, although that this lock library is not robust against PSI. The last one shows the full strength of our semantics in a sense that we can prove specific invariant property on the states, which, we believe, is difficult to achieve using declarative semantics.

In future, we aim to extend our framework to handle other weak consistency models. First, we plan to weaken our assumption of last-write-wins; this can be done by parametrised the

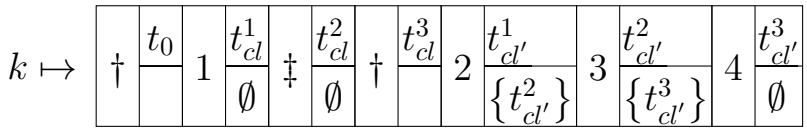
Snapshot function (Def. 4.8). One possibility is to return all versions in the view, instead of the latest one per key, and a transaction is free to read one of them. Another possibility is to choose versions from certain clients cl over others, which can be used to express that these clients cl always have higher priority. However, the question remains if there is other sensible, practical resolution policy than last-write-wins, since almost all commercial distributed key-value stores adopt last-write-wins, where the difference is the definition of relative time.

Second, we plan to weaken the snapshot property, and capture consistency models like *read committed*. For the *read committed* specifically, we can capture it by introducing promises in the style of [Kang et al., 2017], in a sense that, a transaction t may read a version that will be committed by the transaction t' appearing later in the trace. This means that t commits a promised version in the kv-store, and the transaction t' fulfils the promise later. Note that, by doing this, we still ensure that a transaction executes in one reduction step, which has been shown useful in our two applications.

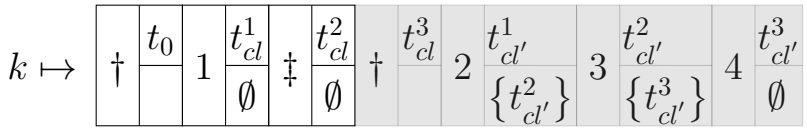
Third, Doherty et al. [2019] proposed an operational semantics on dependency graphs for C11 release and acquire fragment. This fragment is a variant of causal consistency. Thus we believe we can also model this fragment. Furthermore, our semantics may be helpful in term of modelling transactional memory. Separately, in contrast to low level key-value stores, [Burckhardt et al., 2014; Shapiro et al., 2011] on conflict-free data types (CRDTs) focus on abstract data structures, for example, queue. This will be a challenge for us, because we focus on the states, while CRDTs focus on the property of operations. We want to extend our operational semantics to cover these high-level data structures; we can record the abstract states in versions, for example, the states of a queue.

Fourth, our model does not support the allocation and deallocation of keys. This can be achieved by introducing special versions, that identify the aliveness information of keys. For example in Fig. 8.2a, we use † for allocation and ‡ for deallocation, which cuts this list of versions into segments. This affects our views. Once a client view observes † or ‡, it is disallowed to exclude these versions. For example, in the view presented in Fig. 8.2b the † version written by t^3_{cl} must be included for future transactions. Recall that if a consistency model does not satisfy `MR` (in Fig. 4.6a), a client can exclude version after committing a transaction. These special versions also affect the closure check in the execution test, in a sense that a view only needs to be closed with respect to the latest, observable segment. Assume that we want to check if the view presented in Fig. 8.2b is closed with respect to $R_{\mathsf{cc}} = \mathsf{WR} \cup \mathsf{SO}$ relation. This view is clearly not closed by our current $\texttt{PreClosed}\,(\mathcal{K}, u, R_{\mathsf{cc}})$ definition, because the view include the *dagger* version written by t^3_{cl}, but not any previous versions written by the same client cl. However, since *dagger* means allocation, any previous versions before *dagger* are considered as if on a different key, and therefore this view is closed with respect to $R_{\mathsf{cc}} = \mathsf{WR} \cup \mathsf{SO}$ relation.

We plan to validate further the usefulness of our framework, by building verification tools. As the first, we plan to build lightweight tools, for example, implementing our semantics to



(a) Key-value stores with two aliveness segments



(b) An example view on Fig. 8.2a

Figure 8.2: Allocation and deallcation on key-value stores

generate litmus tests for implementations, and do bounded model checking on client programs. In Chapter 6, we provide a full formal verification on implementation protocols. However, the real distributed key-value store implementations are extremely complicated, for example, Eiger [Lloyd et al., 2013], Wren [Spirovska et al., 2018] and Red-Blue [Li et al., 2012], and the codebases of commercial key-value stores, such as Dynamo DB in Amazon Web Service, are often private. Therefore it is impossible to do heavyweight verification on these databases. By contrast, we can generate litmus tests and use them to check if these databases violate the desired consistency model. Once we have an implementation of our semantics, we can use this implementation as an engine to searching traces that distinguish two consistency models. With these traces, we can then construct the programs and the expected result of these programs, as the litmus tests.

In Chapter 7, we manually verify the correctness of small programs. However, we want to explore robustness results for bigger programs such as TPC-C [TPCC, 1992] and RUBiS [RUBIS, 2008], which is not feasible for hand-writing proofs. We plan to build a bounded model checking tool. In contrast to [Gotsman et al., 2016; Zeller, 2017] that builds model checking tools based on declarative semantics, our tool should have better performance, because we check the invariant step by step. We should be able to use this tool to check other general property such as transaction chopping [Shasha et al., 1995; Cerone et al., 2015b], or specific properties, such as that the overall balance remains positive in the bank example discussed in Chapter 7.

Last, we think it will be useful to mechanise our semantics, for example in coq. This will provide high assurance of our semantics, and can be used to generate a verified implementation of our semantics.

Bibliography

Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph.D. Dissertation. Massachusetts Institute of Technology, iambridge, MA, USA.

Jade Alglave. 2010. *A shared Memory Poetics.* Ph.D. Dissertation. l' Université Paris 7 –Denis Diderot.

M. Alomari, M. Cahill, A. Fekete, and U. Rohm. 2008. The Cost of Serializability on Platforms That Use Snapshot Isolation. In *2008 IEEE 24th International Conference on Data Engineering.* 576–585. https://doi.org/10.1109/ICDE.2008.4497466

Amazon. 2019. Amazon Web Service Documentation. https://aws.amazon.com/documentation/

M. Ardekani, P. Sutra, and M. Shapiro. 2013. Non-monotonic Snapshot Isolation: Scalable and Strong Consistency for Geo-replicated Transactional Systems. In *Proceedings of the 32nd Leibniz International Proceedings in Informatics (LIPIcs).* 163–172.

Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. 2014. G-DUR: A Middleware for Assembling, Analyzing, and Improving Transactional Protocols. In *Proceedings of the 15th International Middleware Conference (Middleware'14).* ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/2663165.2663336

Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Scalable Atomic Visibility with RAMP Transactions. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data.* 27–38.

Mark Batty. 2014. *The C11 and C++11 Concurrency Model.* Ph.D. Dissertation. University of Cambridge.

Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC Atomics in C11 and OpenCL. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16).* ACM, New York, NY, USA, 634–648. https://doi.org/10.1145/2837614.2837637

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. 2019. Checking Robustness Against Snapshot Isolation. *CoRR* abs/1905.08406 (2019). arXiv:1905.08406 http://arxiv.org/abs/1905.08406

Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. 2006. PRACTI Replication. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3 (NSDI'06)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=1267680.1267685

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

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

Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1986. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

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

Sebastian Burckhardt. 2014. Principles of Eventual Consistency. *Found. Trends Program. Lang.* 1, 1-2 (October 2014), 1–150. https://doi.org/10.1561/2500000011

Sebastian Burckhardt, Manuel Fahndrich, Daan Leijen, and Mooly Sagiv. 2012. Eventually Consistent Transactions. In *Proceedings of the 21nd European Symposium on Programming*. Springer.

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*. ACM, 271–284.

Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. 2015. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*. 568–590. https://doi.org/10.4230/LIPIcs.ECOOP.2015.568

Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015a. A Framework for Transactional Consistency Models with Atomic Visibility. In *Proceedings of the 26th International Conference on Concurrency Theory (Leibniz International Proceedings in Informatics (LIPIcs))*, Luca Aceto and David de Frutos-Escrig (Eds.), Vol. 42. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 58–71. https://doi.org/10.4230/LIPIcs.CONCUR.2015.58

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

Andrea Cerone, Alexey Gotsman, and Hongseok Yang. 2015b. Transaction Chopping for Parallel Snapshot Isolation. In *Proceedings of the 29th International Symposium on Distributed Computing*. 388–404.

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

Nathan Chong, Tyler Sorensen, and John Wickerson. 2018. The Semantics of Transactions and Weak Memory in x86, Power, ARM, and C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 211–225. https://doi.org/10.1145/3192366.3192373

Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. 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)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/3087801.3087802

Khuzaima Daudjee and Kenneth Salem. 2006. Lazy Database Replication with Snapshot Isolation. In *Proceedings of the 32Nd International Conference on Very Large Data Bases (VLDB '06)*. VLDB Endowment, 715–726. http://dl.acm.org/citation.cfm?id=1182635.1164189

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels.

2007. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.* 41, 6 (October 2007), 205–220.

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

Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. 2013. Towards formally specifying and verifying transactional memory. *Formal Asp. Comput.* 25, 5 (2013), 769–799. https://doi.org/10.1007/s00165-012-0225-8

Brijesh Dongol, Radha Jagadeesan, and James Riely. 2018. Transactions in relaxed memory architectures. *Proc. ACM Program. Lang.* 2, POPL (2018), 18:1–18:29. https://doi.org/10.1145/3158106

Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. 2013. 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)*. IEEE Computer Society, Washington, DC, USA, 173–184. https://doi.org/10.1109/SRDS.2013.26

Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. 2005. Database Replication Using Generalized Snapshot Isolation. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS '05)*. IEEE Computer Society, Washington, DC, USA, 73–84. https://doi.org/10.1109/RELDIS.2005.14

K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (November 1976), 624–633. https://doi.org/10.1145/360363.360369

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

C. J. Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference* 10, 1 (1988), 56–66. http://sky.scitech.qut.edu.au/~fidgec/Publications/fidge88a.pdf

Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News* 33, 2 (June 2002), 51–59. https://doi.org/10.1145/564585.564601

Google. 2019. Google Cloud Platform Documentation. https://cloud.google.com/docs/

Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. '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*, Rastislav Bodik and Rupak Majumdar (Eds.). ACM, 371–384.

J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. 1988. Readings in Database Systems. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Chapter Granularity of Locks and Degrees of Consistency in a Shared Data Base, 94–121. http://dl.acm.org/citation.cfm?id=48751.48758

Rachid Guerraoui and Michal Kapalka. 2008. On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*. ACM, New York, NY, USA, 175–184. https://doi.org/10.1145/1345206.1345233

Theo Haerder and Andreas Reuter. 1983. Principles of Transaction-oriented Database Recovery. *Comput. Surveys* 15, 4 (December 1983), 287–317. https://doi.org/10.1145/289.291

P. W. Hutto and M. Ahamad. 1990. Slow memory: weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of 10th International Conference on Distributed Computing Systems*. 302–309. https://doi.org/10.1109/ICDCS.1990.89297

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

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

Eric Koskinen and Matthew Parkinson. 2015. The Push/Pull Model of Transactions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, 186–195.

Eric Koskinen, Matthew Parkinson, and Maurice Herlihy. 2010. Coarse-grained Transactions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 19–30. https://doi.org/10.1145/1706299.1706304

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming Release-acquire Consistency. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 649–662. https://doi.org/10.1145/2837614.2837643

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 618–632. https://doi.org/10.1145/3062341.3062352

Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565.

Mohsen Lesani. 2014. *On the Correctness of Transactional Memory Algorithms*. Ph.D. Dissertation. University of California, Los Angeles.

Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*. 265–278.

Si Liu, Peter Csaba Ölveczky, Keshav Santhanam, Qi Wang, Indranil Gupta, and José Meseguer. 2018. ROLA: A New Distributed Transaction Protocol and Its Formal Analysis. In *Fundamental Approaches to Software Engineering*, Alessandra Russo and Andy Schürr (Eds.). Springer, Cham, 77–93.

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. ACM, 401–416.

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

Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Proceedings of Workshop on Parallel and Distributed Algorithms*. North-Holland, 215–226.

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

Christos Papadimitriou. 1986. *The Theory of Database Concurrency Control*. Computer Science Press, Inc., New York, NY, USA.

Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4 (October 1979), 631–653. https://doi.org/10.1145/322154.322158

Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. 1997. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM, New York, NY, USA, 288–301. https://doi.org/10.1145/268998.266711

Azalea Raad, Ori Lahav, and Viktor Vafeiadis. 2018. On Parallel Snapshot Isolation and Release/Acquire Consistency. In *Proceedings of the 27th European Symposium on Programming*, Amal Ahmed (Ed.). Lecture Notes in Computer Science, Cham, 940–967.

RUBIS 2008. The RUBiS benchmark. https://rubis.ow2.org/index.html.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS'11)*. Springer-Verlag, 386–400.

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

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

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

Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS '94)*. IEEE Computer Society, Washington, DC, USA, 140–149. http://dl.acm.org/citation.cfm?id=645792.668302

TPCC 1992. The TPC-C benchmark. http://www.tpc.org/tpcc/.

Werner Vogels. 2009. Eventually Consistent. *Commun. ACM* 52, 1 (January 2009), 40–44.

Peter Zeller. 2017. Testing Properties of Weakly Consistent Programs with Repliss. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC'17)*. ACM, New York, NY, USA, Article 3, 5 pages. https://doi.org/10.1145/3064889.3064893

Appendix A

Auxiliary Proofs

A.1 Proofs for Section 4.2 (Operational Semantics)

Proposition A.1 (Well-defined fingerprint combination operation)**.** *Given a well-formed fingerprint $\mathcal{F}$ and an operation $o \in \textsc{Op}$, the new fingerprint $\mathcal{F} \lll o$ is a well-formed fingerprint.*

Proof. The operation o may be a read or a write.

(1) [**Case**: $o = (\mathtt{R}, k, v)$] If there is a entry for the key k, that is, $(l, k, v') \in \mathcal{F}$ for some $l \in \{\mathtt{R}, \mathtt{W}\}$ and value v', then the new fingerprint $\mathcal{F} \lll o = \mathcal{F}$ is trivially well-formed. Otherwise, there is no entry for k and the new fingerprint $\mathcal{F} \lll o = \mathcal{F} \uplus (\mathtt{R}, k, v)$ is also well-formed.

(2) [**Case**: $o = (\mathtt{W}, k, v)$] Let $\mathcal{F}' = (\mathcal{F} \setminus \{(\mathtt{W}, k, v') \mid v' \in \textsc{Value}\})$. By the definition of $\lll$, we have $\mathcal{F} \lll o = \mathcal{F}' \uplus \{(\mathtt{W}, k, v)\}$. Since fingerprint $\mathcal{F}'$ contains no write operation for key k, $(\mathtt{W}, k, v') \notin \mathcal{F}'$ for all values v', the new fingerprint $\mathcal{F}' \uplus \{(\mathtt{W}, k, v)\}$ is a well-formed fingerprint. ■

Theorem 4.21 (Well-defined $\mathsf{UpdateKV}$)**.** *Given a well-formed kv-store $\mathcal{K} \in \mathit{KVS}$, a view on the kv-store $u \in \textsc{ViewOn}(\mathcal{K})$, a well-formed fingerprint $\mathcal{F} \in \textsc{Fp}$ and a fresh transaction identifier $t \in \mathsf{NextTxID}(\mathcal{K}, cl)$ for a client cl, the new kv-store $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t)$ is a uniquely defined and well-formed kv-store.*

Proof. It is easy to see that $\mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t)$ is always defined as $\mathcal{F}$ contains finite number of operations. To prove the uniqueness of $\mathcal{K}'$, it suffices to prove that the order in which the effect of operations is propagated to $\mathcal{K}$ is irrelevant, that is, two operations can swap. Suppose $\mathcal{F} = (\mathcal{F}' \uplus \{o\}) \uplus \{o'\}$ for a fingerprint $\mathcal{F}'$ and two operations o, o'. We prove the following result

$$\mathsf{UpdateKV}(\mathcal{K}, u, (\mathcal{F}' \uplus \{o\}) \uplus \{o'\}, t) = \mathsf{UpdateKV}(\mathcal{K}, u, (\mathcal{F}' \uplus \{o'\}) \uplus \{o\}, t) \tag{A.1}$$

(1) [**Case**: $o = (\mathtt{R}, k, v)$ **and** $o' = (\mathtt{R}, k', v')$] Because $\mathcal{F}$ contains at most one read per key, then $k \neq k'$. Let indexes $i = \mathsf{Max}_{<}(u(k))$ and $i' = \mathsf{Max}_{<}(u(k'))$, versions $(v, t', T) = \mathcal{K}(k, i)$ and $(v', t'', T') = \mathcal{K}(k', i')$. Let the new version lists $\mathcal{V} = \mathcal{K}(k)[i \mapsto (v, t', T \cup \{t\})]$ and $\mathcal{V}' = \mathcal{K}(k')[i \mapsto (v', t'', T' \cup \{t\})]$. Because $k \neq k'$ and $t \notin \mathcal{K}$, it is easy to see $\mathcal{K}[k \mapsto \mathcal{V}][k' \mapsto \mathcal{V}'] = \mathcal{K}[k' \mapsto \mathcal{V}'][k \mapsto \mathcal{V}]$, which implies Eq. (A.1).

(2) [**Case**: $o = (\mathtt{W}, k, v)$ **and** $o' = (\mathtt{W}, k', v')$] Because $\mathcal{F}$ contains at most one write per key, then $k \neq k'$. The proof for Eq. (A.1) is similar to step (1).

(3) [**Case**: $o = (\mathtt{R}, k, v)$ **and** $o' = (\mathtt{W}, k', v')$] Noting k and k' may be the same key. If $k \neq k'$, the proof for Eq. (A.1) is similar to step (1). Consider $k = k'$. Let index $i = \mathsf{Max}_{<}(u(k))$, version list $\mathcal{V} = \mathcal{K}(k)$ and version $(v, t', T) = \mathcal{V}(i)$. Since $u \in \textsc{ViewOn}(\mathcal{K})$, the index idx must be in bound, that is, $0 \leq i < |\mathcal{K}(k)|$, therefore $(\mathcal{V}[i \mapsto (v, t', T \cup \{t\})]) :: [(v', t, \emptyset)] = (\mathcal{V} :: [(v', t, \emptyset)])[i \mapsto (v, t', T \cup \{t\})]$, which implies Eq. (A.1).

(4) [**Case**: $o = (\mathtt{W}, k, v)$ **and** $o' = (\mathtt{R}, k', v')$] It is similar to step (3).

We now prove the kv-store $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t)$ is well-formed by showing the following result,

$$\begin{aligned}
&\mathtt{WfKvs}(\mathcal{K}) \wedge \forall t' \in \mathcal{K}.\, (t, t') \notin \mathsf{SO} \wedge \forall k \in \textsc{Key}.\, \forall v \in \textsc{Value}.\, \forall cl \in \mathrm{CID}. \\
&\qquad ((\mathtt{R}, k, v) \in \mathcal{F} \Rightarrow \forall i'.\, t \notin \mathsf{ReadersOf}(\mathcal{K}(k, i'))) \\
&\qquad \wedge ((\mathtt{W}, k, v) \in \mathcal{F} \Rightarrow \forall i'.\, t \neq \mathsf{WriterOf}(\mathcal{K}(k, i'))) \\
&\qquad\qquad \wedge \forall i \in u(k).\, t \neq \mathsf{WriterOf}(\mathcal{K}(k, i)) \Rightarrow \mathtt{WfKvs}(\mathcal{K}') \qquad \text{(A.2)}
\end{aligned}$$

Note that for any fresh transaction identifier t picked by $t \in \mathsf{NextTxID}(\mathcal{K}, cl)$, it implies the hypothesis of Eq. (A.2). We prove Eq. (A.2) by induction on the size of $\mathcal{F}$.

(1) [**Base Case**: $|\mathcal{F}| = \mathbf{0}$] In this case, we know $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, u, \emptyset, t) = \mathcal{K}$, therefore $\mathtt{WfKvs}(\mathcal{K}')$.

(2) [**Inductive Case**: $|\mathcal{F}| > 0$] The next operation may be a read or a write operation.

(i) [**Case**: $\mathcal{F} = \mathcal{F}' \uplus (\mathtt{R}, k, v)$] Let index $i = \mathsf{Max}_{<}(u(k))$, old version $(v, t', T) = \mathcal{K}(k, i)$, and new version list $\mathcal{V} = \mathcal{K}(k)[i \mapsto (v, t', T \uplus \{t\})]$. The intermediate kv-store $\mathcal{K}^*$ is defined by $\mathcal{K}^* = \mathcal{K}[k \mapsto \mathcal{V}]$. Since the original kv-store $\mathcal{K}$ satisfies Eq. (A.2), the fresh transaction identifier $t \notin \mathsf{ReadersOf}(\mathcal{K}(k, i'))$ for all i' such that $0 \leq i' < |\mathcal{K}(k)|$, and therefore $\mathcal{K}^*$ satisfies the well-formed condition Eq. (4.1) in Def. 4.5; because $(t, t') \notin \mathsf{SO}$ for any $t' \in \mathcal{K}$, (It is only possible $(t', t) \in \mathsf{SO}$ for some $t' = \mathcal{K}(k, i')$ where $i' \in u(k)$), and $t \neq \mathsf{WriterOf}(\mathcal{K}(k'', i''))$ for k'', i'' such that $i'' \in u(k'')$, therefore $\mathcal{K}^*$ satisfies Eq. (4.3). Eqs. (4.2) and (4.4) are trivially true for $\mathcal{K}^*$. We proved the intermediate kv-store $\mathcal{K}^*$ is well-formed, $\mathtt{WfKvs}(\mathcal{K}^*)$. As $\mathcal{F}$ is well-formed,

it follows that $(\mathtt{R}, k, v') \notin \mathcal{F}'$, which means that $\mathcal{K}^*, \mathcal{F}', u, t$ satisfy the invariant Eq. (A.2). By inductive hypothesis the final kv-store $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}^*, u, \mathcal{F}', t)$ is a well-formed kv-store.

(ii) [**Case**: $\mathcal{F} = \mathcal{F}' \uplus (\mathtt{W}, k, v)$] Let new version list $\mathcal{V} = \mathcal{K}(k)::[(v, t, \emptyset)]$. The intermediate kv-store $\mathcal{K}^*$ is defined by $\mathcal{K}^* = \mathcal{K}[k \mapsto \mathcal{V}]$. Since the original kv-store $\mathcal{K}$ satisfies Eq. (A.2), the fresh identifier $t \neq \mathsf{WriterOf}(\mathcal{K}(k, i'))$ for all i' such that $0 \leq i' < |\mathcal{K}(k)|$, therefore $\mathcal{K}^*$ satisfies Eq. (4.1); because $(t, t') \notin \mathsf{SO}$ for any $t' \in \mathcal{K}$ (It is only possible $(t', t) \in \mathsf{SO}$ for some t' in $\mathcal{K}(k)$), and t wrote the last version for k in $\mathcal{K}^*$, therefore $\mathcal{K}^*$ satisfies Eq. (4.4) Eqs. (4.2) and (4.3) are trivially true for $\mathcal{K}^*$. We proved the intermediate kv-store is well-fumed, $\mathtt{WfKvs}(\mathcal{K}^*)$. As $\mathcal{F}$ is well-formed, it follows $(\mathtt{W}, k, v') \notin \mathcal{F}'$, which means that $\mathcal{K}^*, \mathcal{F}', u, t$ satisfy the invariant Eq. (A.2). By inductive hypothesis the final kv-store $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}^*, u, \mathcal{F}', t)$ is a well-formed kv-store. ■

Theorem 4.30 (Equivalent normal ET-traces)**.** *For any $\tau \in \mathit{ETTraces}$, there exists an equivalent normalised trace τ^*, that is, $\tau \simeq \tau^*$ and* $\mathtt{NormalisedTrace}(\tau^*)$.

Proof. Let initially $\tau^* = \tau$; we alters the trace until τ^* is a normal trace.

(1) Consider the last step ι for each client cl such that

$$\tau^* = \tau' \xrightarrow{\iota}_{\mathtt{ET}} \tau'' \wedge \iota = (cl, \alpha) \wedge \forall cl'.\, \tau'' = \cdots \xrightarrow{(cl', \_)}_{\mathtt{ET}} \cdots \Rightarrow cl \neq cl'$$

for two trace segments τ', τ''. Let view $u = \mathsf{Last}(\tau')_{|2}(cl)$. If ι is a view shift step, i.e. $\alpha = u'$ for some view u' such that $u \sqsubseteq u'$, we delete this step and re-assign the configurations in τ'' resulting a new segment $\tau^\dagger$ such that

$$\forall i \in \mathbb{N}.\, \forall \mathcal{K} \in \mathrm{KVS}.\, \forall \mathcal{U} \in \textsc{ViewEnvOn}(\mathcal{K})\,.\, \tau''_{|i} = (\mathcal{K}, \mathcal{U}) \Leftrightarrow \tau^\dagger_{|i} = (\mathcal{K}, \mathcal{U}[cl \mapsto u]).$$

We rename the new trace as τ^* and go back step (1). Because trace τ^* has finite steps, step (1) must terminate with trace τ^* that satisfies:

$$\forall \iota \in \textsc{Labels}.\, \forall cl, cl' \in \mathrm{CID}.\, \forall \alpha.\, \forall \tau', \tau''.$$
$$\left(\tau^* = \tau' \xrightarrow{\iota}_{\mathtt{ET}} \tau'' \wedge \iota = (cl, \alpha) \wedge \tau'' = \cdots \xrightarrow{(cl', \_)}_{\mathtt{ET}} \cdots \Rightarrow cl \neq cl'\right) \Rightarrow \alpha \in \textsc{Fp}. \quad \text{(A.3)}$$

(2) Given a trace τ^* satisfying Eq. (A.3), consider the first view-shift step ι for a client cl that is not followed by a step for the same client cl: $\tau^* = \tau' \xrightarrow{\iota}_{\mathtt{ET}} \Gamma \xrightarrow{\iota'}_{\mathtt{ET}} \tau''$ such that $\iota = (cl, u)$ and $\iota' = (cl', \alpha)$ for some α, view u and some clients cl, cl' with $cl \neq cl'$. By Lemma A.2 we can move step ι to the right resulting $\tau' \xrightarrow{\iota'}_{\mathtt{ET}} \Gamma^* \xrightarrow{\iota}_{\mathtt{ET}} \tau''$ for some Γ^*, until the immediate next step is for the same client cl. Note that there must be a step for the

same client cl in τ'' by Eq. (A.3). We rename the new trace as τ^* and go back to step (2). Because there are only finite steps and the number of out-of-order view-shifts decreases after each iteration, step (2) must terminate with trace τ^* such that

$$\begin{aligned}&\forall \iota, \iota' \in \textsc{Labels}.\, \forall cl \in \text{CID}.\, \forall u \in \textsc{View}.\, \forall \tau', \tau''.\\ &\quad \tau^* = \tau' \xrightarrow{\iota}_{\texttt{ET}} \Gamma \xrightarrow{\iota'}_{\texttt{ET}} \tau'' \wedge \iota = (cl, u) \Rightarrow \iota' = (cl, \_)\,. \end{aligned} \tag{A.4}$$

(3) Given a trace τ^* satisfying Eqs. (A.3) and (A.4), consider the two adjacent view-shifts steps ι, ι' such that $\tau^* = \tau' \xrightarrow{\iota}_{\texttt{ET}} \Gamma \xrightarrow{\iota'}_{\texttt{ET}} \tau''$ where $\iota = (cl, u)$ and $\iota' = (cl, u')$ for two views u, u'. By Lemma A.3, we can merge these two steps resulting $\tau' \xrightarrow{\iota'}_{\texttt{ET}} \tau''$ and go back to step (3). Because there are only finite steps and the number of two adjacent view-shifts decreases after each iteration, step (3) must terminate with trace τ^* such that

$$\begin{aligned}&\forall \iota, \iota' \in \textsc{Labels}.\, \forall cl \in \text{CID}.\, \forall u \in \textsc{View}.\, \forall \tau', \tau''.\\ &\quad \tau^* = \tau' \xrightarrow{\iota}_{\texttt{ET}} \Gamma \xrightarrow{\iota'}_{\texttt{ET}} \tau'' \wedge \iota = (cl, u) \Rightarrow \exists \mathcal{F} \in \textsc{Fp}.\, \iota' = (cl, \mathcal{F})\,. \end{aligned} \tag{A.5}$$

(4) Last, given a trace τ^* satisfying Eqs. (A.3) to (A.5), consider an update step ι' without a view-shift predecessor, that is, $\tau^* = \tau' \xrightarrow{\iota}_{\texttt{ET}} \Gamma \xrightarrow{\iota'}_{\texttt{ET}} \tau''$ such that $\iota = (cl, \mathcal{F})$ and $\iota' = (cl', \mathcal{F}')$ for some fingerprints $\mathcal{F}, \mathcal{F}'$ and clients cl, cl' with $cl \neq cl'$. We inject an identity view shift in between resulting $\tau' \xrightarrow{\iota}_{\texttt{ET}} \Gamma \xrightarrow{(cl, \Gamma_{|2}(cl))}_{\texttt{ET}} \Gamma \xrightarrow{\iota'}_{\texttt{ET}} \tau''$. Rename the new trace as τ^* and go back to step (4). Because there are only finite steps, step (4) must terminate with trace τ^* satisfying

$$\begin{aligned}&\forall \iota, \iota' \in \textsc{Labels}.\, \forall cl \in \text{CID}.\, \forall \mathcal{F} \in \textsc{Fp}.\, \forall \tau', \tau''.\\ &\quad \tau^* = \tau' \xrightarrow{\iota}_{\texttt{ET}} \Gamma \xrightarrow{\iota'}_{\texttt{ET}} \tau'' \wedge \iota' = (cl, \mathcal{F}) \Rightarrow \exists u \in \textsc{View}.\, \iota' = (cl, u)\,. \end{aligned} \tag{A.6}$$

Now we have an equivalent trace τ^* such that $\tau \simeq \tau^*$ and Eqs. (A.5) and (A.6) imply $\texttt{NormalisedTrace}(\tau^*)$. ■

Lemma A.2 (View-shift right move)**.** *Given a trace $\tau \in \mathit{ETTraces}(\texttt{ET})$ for some $\texttt{ET}$, and a view-shift step (cl, u) for a client cl in the trace τ, if the view-shift is not followed by a step for the same client cl, this view-shift can be moved right without changing the final configuration.*

$$\begin{aligned}&\forall cl' \in \mathit{CID}.\, \forall \iota, \iota' \in \mathit{Labels}.\, \forall \alpha, \tau', \tau''.\\ &\quad \tau = \tau' \xrightarrow{\iota}_{\texttt{ET}} \Gamma \xrightarrow{(cl,u)}_{\texttt{ET}} \Gamma' \xrightarrow{(cl',\alpha)}_{\texttt{ET}} \Gamma'' \xrightarrow{\iota'}_{\texttt{ET}} \tau'' \wedge cl \neq cl'\\ &\quad \Rightarrow \exists \Gamma^* \in \mathit{Conf}.\, \tau' \xrightarrow{\iota}_{\texttt{ET}} \Gamma \xrightarrow{(cl',\alpha)}_{\texttt{ET}} \Gamma^* \xrightarrow{(cl,u)}_{\texttt{ET}} \Gamma'' \xrightarrow{\iota'}_{\texttt{ET}} \tau''. \end{aligned} \tag{A.7}$$

Proof. Let configuration $(\mathcal{K}, \mathcal{U}) = \Gamma$; we preform case analysis on α.

(1) [**Case**: $\alpha = u'$] Given the two view-shifts for clients cl and cl' respectively, we know the

configuration Γ'' is given by $\Gamma'' = (\mathcal{K}, \mathcal{U}\,[cl \mapsto u]\,[cl' \mapsto u'])$. Since two clients are distinct $cl \neq cl'$, then we have $\Gamma'' = (\mathcal{K}, \mathcal{U}\,[cl' \mapsto u']\,[cl \mapsto u])$; therefore we prove Eq. (A.7) by picking $\Gamma^* = (\mathcal{K}, \mathcal{U}\,[cl' \mapsto u'])$.

(2) [**Case**: $\alpha = \mathcal{F}$] We know $\Gamma' = (\mathcal{K}, \mathcal{U}\,[cl \mapsto u])$ and $\Gamma'' = (\mathcal{K}', \mathcal{U}\,[cl \mapsto u]\,[cl' \mapsto u'])$ for some view u', kv-store $\mathcal{K}' = \mathsf{UpdateKV}\,(\mathcal{K}, \mathcal{U}\,(cl'), \mathcal{F}, t)$ with a fresh transition identifier t, and $(\mathcal{K}, \mathcal{U}\,(cl'))\ \xrightarrow{\mathcal{F}}_{\mathtt{ET}}\ (\mathcal{K}', u')$. Let $\Gamma^* = (\mathcal{K}', \mathcal{U}\,[cl' \mapsto u'])$ which gives us Eq. (A.7). ■

Lemma A.3 (View-shift absorption)**.** *Given a trace* $\tau \in \mathit{ETTraces}\,(\mathtt{ET})$ *under* $\mathtt{ET}$*, two adjacent view-shifts* $(cl, u), (cl, u')$ *for a client* cl *in the trace can be merged,*

$$\forall \tau', \tau''.\, \forall \iota, \iota' \in \mathit{Labels}.\\ \tau = \tau' \xrightarrow{\iota}_{\mathtt{ET}} \Gamma \xrightarrow{(cl,u)}_{\mathtt{ET}} \Gamma' \xrightarrow{(cl,u')}_{\mathtt{ET}} \Gamma'' \xrightarrow{\iota'}_{\mathtt{ET}} \tau'' \Rightarrow \tau' \xrightarrow{\iota}_{\mathtt{ET}} \Gamma \xrightarrow{(cl,u')}_{\mathtt{ET}} \Gamma'' \xrightarrow{\iota'}_{\mathtt{ET}} \tau''. \tag{A.8}$$

Proof. Let configuration $(\mathcal{K}, \mathcal{U}) = \Gamma$; it is easy to see $\Gamma = (\mathcal{K}, \mathcal{U}\,[cl \mapsto u])$, $\Gamma = (\mathcal{K}, \mathcal{U}\,[cl \mapsto u'])$ and $\mathcal{U}\,(cl) \sqsubseteq u \sqsubseteq u'$, which implies Eq. (A.8). ■

Theorem 4.31 (Equivalent expressibility)**.** *For any* $\mathtt{ET} \in \mathit{ExecutionTest}$*,* $\mathsf{ConsisModel}\,(\mathtt{ET}) = \bigcup_{\mathtt{P} \in \mathit{PROGS}} [\![\mathtt{P}]\!]_{\mathtt{ET}}$*, where* $\mathsf{ConsisModel}$ *is defined in Def. 4.27 and* $[\![\mathtt{P}]\!]_{\mathtt{ET}}$ *is defined in Def. 4.24.*

Proof. We prove $\mathsf{ConsisModel}\,(\mathtt{ET}) \supseteq \bigcup_{\mathtt{P} \in \mathrm{PROGS}} [\![\mathtt{P}]\!]_{\mathtt{ET}}$ and $\mathsf{ConsisModel}\,(\mathtt{ET}) \subseteq \bigcup_{\mathtt{P} \in \mathrm{PROGS}} [\![\mathtt{P}]\!]_{\mathtt{ET}}$ respectively.

(1) [**Case**: $\bigcup_{\mathtt{P} \in \mathbf{Progs}} [\![\mathtt{P}]\!]_{\mathtt{ET}} \subseteq \mathsf{ConsisModel}\,(\mathtt{ET})$] By definition of $\bigcup_{\mathtt{P} \in \mathrm{PROGS}} [\![\mathtt{P}]\!]_{\mathtt{ET}}$ It is sufficient to prove that for any trace η, initial configuration $(\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}_0)$ and final configuration $(\mathcal{K}, \mathcal{U}, \mathcal{E})$ with program $\mathtt{P}'$,

$$\mathsf{Dom}\,(\mathtt{P}) \subseteq \mathsf{Dom}\,(\mathcal{E}) \wedge \eta = (\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}_0), \mathtt{P} \Rightarrow^*_{\mathtt{ET}} (\mathcal{K}, \mathcal{U}, \mathcal{E}), \mathtt{P}' \\ \Rightarrow (\mathcal{K}_0, \mathcal{U}_0) \Rightarrow^*_{\mathtt{ET}} (\mathcal{K}', \mathcal{U}'). \tag{A.9}$$

We prove Eq. (A.9) by induction on the length of the trace η.

(i) [**Base Case**: $\eta = (\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}_0), \mathtt{P}$] Eq. (A.9) trivially holds.

(ii) [**Inductive Case**: $\eta = (\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}_0), \mathtt{P} \Rightarrow^n_{\mathtt{ET}} (\mathcal{K}', \mathcal{U}', \mathcal{E}'), \mathtt{P}'' \xrightarrow{\iota}_{\mathtt{ET}} (\mathcal{K}, \mathcal{U}, \mathcal{E}), \mathtt{P}'$] By inductive hypothesis, there must exist a $\mathtt{ET}$-trace τ such that $\tau = (\mathcal{K}_0, \mathcal{U}_0) \Rightarrow^*_{\mathtt{ET}} (\mathcal{K}', \mathcal{U}')$. We append steps to $\mathtt{ET}$-trace τ depending on the step ι. (a) [**Case**: $\iota = (cl, \bullet)$] This means $\mathcal{K}' = \mathcal{K}''$ and $\mathcal{U}' = \mathcal{U}''$, and we immediately have the proof for Eq. (A.9) with $\mathtt{ET}$-trace τ. (b) [**Case**: $\iota = (cl, u, \mathcal{F})$] This means $\mathcal{K}' = \mathsf{UpdateKV}\,(\mathcal{K}'', u, \mathcal{F}, t)$ and $\mathcal{U}' = \mathcal{U}''\,[cl \mapsto u']$ for a fresh t and a view u' such that $(\mathcal{K}'', u)\ \xrightarrow{\mathcal{F}}_{\mathtt{ET}}\ (\mathcal{K}', u')$. Let the new $\mathtt{ET}$-trace τ' be $\tau' = \tau \xrightarrow{(cl,u)}_{\mathtt{ET}} (\mathcal{K}'', \mathcal{U}''\,[cl \mapsto u]) \xrightarrow{(cl,\mathcal{F})}_{\mathtt{ET}} (\mathcal{K}', \mathcal{U}')$ which implies Eq. (A.9).

(2) [**Case**: $\mathsf{ConsisModel}\,(\mathtt{ET}) \subseteq \bigcup_{\mathtt{P}\in\mathbf{Progs}} [\![\mathtt{P}]\!]_{\mathtt{ET}}$] Since there exist equivalent normalised traces for all traces in $\mathsf{ConsisModel}\,(\mathtt{ET})$ by Theorem 4.30, it is sufficient to prove that for any trace $\tau \in \mathsf{ConsisModel}\,(\mathtt{ET})$, initial configuration $(\mathcal{K}_0, \mathcal{U}_0)$ and final configuration $(\mathcal{K}, \mathcal{U})$,

$$\begin{aligned}
&\mathtt{NormalisedTrace}\,(\tau) \wedge \tau = (\mathcal{K}_0, \mathcal{U}_0) \Rightarrow^*_{\mathtt{ET}} (\mathcal{K}, \mathcal{U}) \Rightarrow \\
&\qquad \exists \mathcal{E}_0, \mathcal{E} \in \textsc{ClientEnvs}.\ \exists \mathtt{P}, \mathtt{P}' \in \textsc{Progs}. \\
&\mathsf{Dom}\,(\mathtt{P}) = \mathsf{Dom}\,(\mathcal{E}_0) \wedge (\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}_0)\,, \mathtt{P} \Rightarrow^*_{\mathtt{ET}} (\mathcal{K}, \mathcal{U}, \mathcal{E})\,, \mathtt{P}' \\
&\qquad\qquad \wedge \forall cl \in \mathsf{Dom}\,(\mathtt{P}')\,.\, \mathtt{P}'(cl) = \mathtt{skip}.
\end{aligned} \tag{A.10}$$

We prove Eq. (A.10) by induction on the length of the trace τ. Note that the number of steps in trace τ must be an *even* number.

(i) [**Base Case**: $\tau = (\mathcal{K}_0, \mathcal{U}_0)$] We pick $\mathtt{P} = \mathcal{U}_0 = \emptyset$ and $\eta = (\mathcal{K}_0, \mathcal{U}_0, \mathcal{U}_0)\,, \mathtt{P}$ that implies Eq. (A.10).

(ii) [**Inductive Case**: $\tau = (\mathcal{K}_0, \mathcal{U}_0) \Rightarrow^n_{\mathtt{ET}} (\mathcal{K}', \mathcal{U}') \xrightarrow{(cl,u)}_{\mathtt{ET}} (\mathcal{K}', \mathcal{U}'\,[cl \mapsto u]) \xrightarrow{(cl,\mathcal{F})}_{\mathtt{ET}} (\mathcal{K}, \mathcal{U})$] By inductive hypothesis, there must be a program trace η such that

$$\eta = (\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}_0)\,, \mathtt{P} \Rightarrow^*_{\mathtt{ET}} (\mathcal{K}', \mathcal{U}', \mathcal{E}')\,, \mathtt{P}' \wedge \forall cl \in \mathsf{Dom}\,(\mathtt{P}')\,.\, \mathtt{P}'(cl) = \mathtt{skip}.$$

for client environment $\mathcal{E}_0, \mathcal{E}'$ and program $\mathtt{P}'$. We now construct a new initial by extending a new transaction for client cl; this transaction has the fingerprint $\mathcal{F}$. Recall that a well-formed fingerprint contains at most one read and one write for each key. We define the transactional command for fingerprint $\mathcal{F}$, written $\mathsf{TransFp}\,(\mathcal{F})$ as the following:

$$\begin{aligned}
\mathsf{TransFp}\,(\emptyset) &\stackrel{def}{=} \mathtt{skip}, \\
\mathsf{TransFp}\,(\mathcal{F} \uplus (\mathtt{R}, k, v)) &\stackrel{def}{=} \mathtt{x} := \mathtt{rd}\,(k)\ ;\ \mathsf{TransFp}\,(\mathcal{F})\,, \\
\mathsf{TransFp}\,(\mathcal{F} \uplus (\mathtt{W}, k, v)) &\stackrel{def}{=} \mathtt{wt}\,(k, v)\ ;\ \mathsf{TransFp}\,(\mathcal{F}) \\
&\qquad\qquad \text{where } \forall k' \in \textsc{Key}.\ \forall v' \in \textsc{Value}.\ (\mathtt{R}, k', v') \notin \mathcal{F}.
\end{aligned}$$

We then define a function that extends the command for a client:

$$\mathsf{ExtendProgram}\,(\mathtt{P}, cl, \mathtt{C}) \stackrel{def}{=} \begin{cases} \mathtt{P}\,[cl \mapsto (\mathtt{C}'\ ;\ \mathtt{C})] & \text{if } \mathtt{P}\,(cl) = \mathtt{C}' \\ \mathtt{P} \uplus \{cl \mapsto \mathtt{C}\} & \text{otherwise} \end{cases}$$

Let $\mathtt{T} = \mathsf{TransFp}\,(\mathcal{F})$ be transactional command $\mathsf{ExtendProgram}\,(\mathtt{P}', cl, [\ \mathtt{T}\ ])$ be the new initial program, and $\mathtt{P}'' = \mathsf{ExtendProgram}\,(\mathtt{P}', cl, [\ \mathtt{T}\ ])$ be the new final program, and consequently apply $\mathsf{ExtendProgram}$ to all intermediate programs. This means,

we have a new trace η' such that

$$\eta' = (\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}_0), \mathsf{ExtendProgram}(\mathtt{P}, cl, [\ \mathtt{T}\ ]) \Rightarrow^*_{\mathtt{ET}} (\mathcal{K}', \mathcal{U}', \mathcal{E}'), \mathtt{P}''$$
$$\wedge \forall cl' \in \mathsf{Dom}(\mathtt{P}'') . (cl = cl' \Rightarrow \mathtt{P}''(cl') = [\ \mathtt{T}\ ]) \wedge (cl \neq cl' \Rightarrow \mathtt{P}''(cl') = \mathtt{skip}).$$

The only next step for η' is to execute the [T] for cl. Recall that

$$(\mathcal{K}', \mathcal{U}'[cl \mapsto u]) \xrightarrow{(cl,\mathcal{F})}_{\mathtt{ET}} (\mathcal{K}, \mathcal{U}).$$

Given CAtomicTrans shown in Fig. 4.2, the client local stack is $s = \mathcal{E}'(cl)$ and the initial snapshot σ for T is $\sigma = \mathsf{Snapshot}(\mathcal{K}', u)$. By the hypothesis, we know that $(\mathcal{K}', u) \xrightarrow{\mathcal{F}}_{\mathtt{ET}} (\mathcal{K}', \mathcal{U}(cl))$ and by well-formed condition for ET, we know

$$\forall k \in \textsc{Key}. \forall v \in \textsc{Value}. (\mathtt{R}, k, v) \in \mathcal{F} \Rightarrow \sigma(k) = v,$$

which implies $(s, \sigma, \emptyset), \mathtt{T} \rightsquigarrow^* (s', \sigma', \mathcal{F}), \mathtt{skip}$ for some stack s' and snapshot σ. By $(\mathcal{K}', u) \xrightarrow{\mathcal{F}}_{\mathtt{ET}} (\mathcal{K}', \mathcal{U}(cl))$, $\mathtt{CanCommit}(\mathcal{K}, u, \mathcal{F})$ and $\mathtt{ViewShift}(\mathcal{K}, u, \mathcal{K}', \mathcal{U}(cl))$ must hold. Thus we have a trace such that

$$\eta' \xrightarrow{(cl,u,\mathcal{F})}_{\mathtt{ET}} (\mathcal{K}, \mathcal{U}, \mathcal{E}'[cl \mapsto s']), \mathtt{P}^* \wedge \forall cl \in \mathsf{Dom}(\mathtt{P}^*) . \mathtt{P}^*(cl) = \mathtt{skip}$$

for some program $\mathtt{P}^*$, which implies Eq. (A.10). ■

A.2 Proofs for Section 5.1 (Correspondence to Dependency Graph)

Proposition A.4 (Well-defined KToD)**.** *Given a kv-store $\mathcal{K} \in KVS$, the dependency graph induced by the kv-store $\mathsf{KToD}(\mathcal{K})$ is well-formed, that is, $\mathsf{KToD}(\mathcal{K}) \in DGraphs$.*

Proof. First we prove the dependency graph $\mathsf{KToT}(\mathcal{K})$ match the type constraint of the nodes. Consider a transaction $t \in \mathcal{K}$.

(1) [**Case**: $t = t_0$] By the definition of versions Version, t_0 cannot in any reader set; By well-formedness of $\mathcal{K}$, the initial transaction wrote the first version for each key (Eq. (4.2)); and it only wrote those versions (snapshot property in Eq. (4.1)). Therefore $\mathsf{KToT}(\mathcal{K})(t_0) = \{(\mathtt{W}, k, v_0) \mid k \in \textsc{Key} \wedge v_0 \in \mathsf{InitialValue}(k)\}$.

(2) [**Case**: $t \neq t_0$] By snapshot property of $\mathcal{K}$ (Eq. (4.1)), it is easy to see that $\mathsf{KToT}(\mathcal{K})(t) \in$ Fp.

Let $\mathsf{WR} = \mathsf{WR}_{\mathcal{K}}, \mathsf{WW} = \mathsf{WW}_{\mathcal{K}}, \mathsf{RW} = \mathsf{RW}_{\mathcal{K}}$. Now we prove that the relations are well-formed.

(1) [**Case**: **write-read dependency** WR]

(i) Suppose a transaction t such that $(\mathtt{R}, k, \_) \in \mathsf{KToT}(\mathcal{K})(t)$. There exists a version $\mathcal{K}(k,i)$ for a key k and an index i, such that $t \in \mathsf{ReadersOf}(\mathcal{K}(k,i))$. We have $(\mathsf{WriterOf}(\mathcal{K}(k,i)), t) \in \mathsf{WR}$ which implies Eq. (5.3).

(ii) Suppose transactions t, t' such that $(t, t') \in \mathsf{WR}$. By definition of WR, there exists a version $\mathcal{K}(k,i)$ for a key k and an index i, such that $t = \mathsf{WriterOf}(\mathcal{K}(k,i))$ and $t' \in \mathsf{ReadersOf}(\mathcal{K}(k,i))$, which immediately implies Eq. (5.2).

(iii) Suppose transactions t, t' such that $(t, t') \in \mathsf{WR}$. By definition of WR, there exists a version $\mathcal{K}(k,i)$ for a key k and an index i, such that $t = \mathsf{WriterOf}(\mathcal{K}(k,i))$ and $t' \in \mathsf{ReadersOf}(\mathcal{K}(k,i))$. Because $\mathcal{K}$ is well-formed (Eq. (4.3)), it must be the case that $(t, t') \notin \mathsf{SO}$, which implies Eq. (5.4).

(iv) Suppose transactions t, t', t'' such that $(t', t), (t'', t) \in \mathsf{WR}$. There exist two versions $\mathcal{K}(k,i), \mathcal{K}(k,i')$ for a key k and indices i, i', such that $t \in \mathsf{ReadersOf}(\mathcal{K}(k,i))$ and $t \in \mathsf{ReadersOf}(\mathcal{K}(k,i'))$. Because $\mathcal{K}$ is well-formed (Eq. (4.1)), it must be the case that $i = i'$ which means that $t' = t'' = \mathsf{WriterOf}(\mathcal{K}(k,i))$ and thus Eq. (5.5).

(2) [**Case**: **write-write dependency** WW]

(i) Suppose transactions t, t' such that

$$(\mathtt{W}, k, \_) \in \mathsf{KToT}(\mathcal{K})(t) \text{ and } (\mathtt{W}, k, \_) \in \mathsf{KToT}(\mathcal{K})(t').$$

There exist two versions $\mathcal{K}(k,i), \mathcal{K}(k,i')$ for a key k and indices i, i', such that $t = \mathsf{WriterOf}(\mathcal{K}(k,i))$ and $t' = \mathsf{WriterOf}(\mathcal{K}(k,i'))$. If $i = i'$ then $t = t'$; if $i < i'$ or $i > i'$, then $(t, t') \in \mathsf{WW}$ or $(t', t) \in \mathsf{WW}$ respectively. Thus we prove Eq. (total-WW).

(ii) Suppose transactions t, t' such that $(t, t') \in \mathsf{WW}$. There exist two versions $\mathcal{K}(k,i)$, $\mathcal{K}(k,i')$ for a key k and indices i, i' $t = \mathsf{WriterOf}(\mathcal{K}(k,i))$ and $t' = \mathsf{WriterOf}(\mathcal{K}(k,i'))$, which immediately implies Eq. (5.7).

(iii) Suppose transactions t, t' such that $(t, t') \in \mathsf{WW}$. We prove $t' \neq t_0$ by contradiction. Assume $t' = t_0$. There exist a version $\mathcal{K}(k,i)$ for a key k and an index $i > 0$, such that $t_0 = \mathsf{WriterOf}(\mathcal{K}(k,i))$. However, $t_0 = \mathsf{WriterOf}(\mathcal{K}(k,0))$ by well-formedness; there are two versions for k written by t_0 which contradicts Eq. (4.1). Therefore Eq. (5.8) holds.

(iv) Suppose transactions t, t' such that $(t, t') \in \mathsf{WW}$. There exist two versions $\mathcal{K}(k,i)$, $\mathcal{K}(k,i')$ for a key k and indices i, i', such that $i < i'$, $t = \mathsf{WriterOf}(\mathcal{K}(k,i))$ and $t' = \mathsf{WriterOf}(\mathcal{K}(k,i'))$. Because $\mathcal{K}$ is well-formed (Eq. (4.4)), it must be the case that $(t', t) \notin \mathsf{SO}$, which implies Eq. (5.9).

(v) Suppose transactions t, t' such that $(t, t') \in \mathsf{WW}$. We prove $t \neq t'$ by contradiction. Assume $t = t'$. There exist two versions $\mathcal{K}(k,i), \mathcal{K}(k,i')$ for a key k and indices i, i',

such that $t = \mathsf{WriterOf}(\mathcal{K}(k,i)) = \mathsf{WriterOf}(\mathcal{K}(k,i'))$; this contradicts Eq. (4.1). Similarly, we prove $(t',t) \notin \mathsf{WW}$ by contradiction. Assume $(t',t) \in \mathsf{WW}$, it means $t = \mathsf{WriterOf}(\mathcal{K}(k,i))$, $t' = \mathsf{WriterOf}(\mathcal{K}(k,i'))$ and $i > i'$. However, we also have $(t,t') \in \mathsf{WW}$ and $i < i'$, which leads to a contradiction. Thus Eq. (irref-asym-WW) holds.

(vi) Suppose transactions t, t', t'' such that $(t,t'),(t',t'') \in \mathsf{WW}$. There exist versions $\mathcal{K}(k,i), \mathcal{K}(k,i'), \mathcal{K}(k,i'')$ for a key k and indices i, i', such that $i < i' < i''$, $t = \mathsf{WriterOf}(\mathcal{K}(k,i))$, $t' = \mathsf{WriterOf}(\mathcal{K}(k,i'))$, and $t'' = \mathsf{WriterOf}(\mathcal{K}(k,i''))$. By definition of WW, it must be the case that $(t,t'') \in \mathsf{WW}$, which implies Eq. (transitive-WW).

(3) [**Case**: **read-write anti-dependency** RW] Suppose transactions t, t' such that

$$t \neq t' \wedge (t,t') \in \mathsf{RW}. \tag{A.11}$$

Eq. (A.11) holds iff there a transaction t'' and two versions $\mathcal{K}(k,i), \mathcal{K}(k,i')$ such that $i < i'$, $t \in \mathsf{ReadersOf}(\mathcal{K}(k,i))$ and $t' = \mathsf{WriterOf}(\mathcal{K}(k,i'))$; this means that

$$(\mathsf{WriterOf}(\mathcal{K}(k,i)), t) \in \mathsf{WR} \wedge (\mathsf{WriterOf}(\mathcal{K}(k,i)), t') \in \mathsf{WW} \wedge t \neq t'. \tag{A.12}$$

Since Eq. (A.11) implies Eq. (A.12) and vice versa, we have the proof for Eq. (5.11). ■

Proposition A.5 (Well-defined DToK)**.** *Given a dependency graph $\mathcal{G} \in$ DGraphs, the kv-store induced by dependency graph, $\mathsf{DToK}(\mathcal{G})$, is well-formed, that is, $\mathsf{DToK}(\mathcal{G}) \in KVS$.*

Proof. Let $\mathcal{K} = \mathsf{DToK}(\mathcal{G})$. Given a key k, by the well-formedness definition for kv-store, it is sufficient to prove the following cases.

(1) Suppose two indices i, i' such that $\mathsf{ReadersOf}(\mathcal{K}(k,i)) \cup \mathsf{ReadersOf}(\mathcal{K}(k,i')) \neq \emptyset$. Assume $t \in \mathsf{ReadersOf}(\mathcal{K}(k,i)) \cup \mathsf{ReadersOf}(\mathcal{K}(k,i))$. There must exist edges

$$(\mathsf{WriterOf}(\mathcal{K}(k,i)), t), (\mathsf{WriterOf}(\mathcal{K}(k,i')), t) \in \mathsf{WR}_{\mathcal{G}}.$$

Because $\mathcal{G}$ is well-formed (Eq. (5.5)), $\mathsf{WriterOf}(\mathcal{K}(k,i)) = \mathsf{WriterOf}(\mathcal{K}(k,i'))$ thus $i = i'$.

(2) Suppose two indices i, i' such that $\mathsf{WriterOf}(\mathcal{K}(k,i)) = \mathsf{WriterOf}(\mathcal{K}(k,i'))$. We prove $i = i'$ by contradiction. Assume $i \neq i'$. Without losing generality, this means that $(\mathsf{WriterOf}(\mathcal{K}(k,i)), \mathsf{WriterOf}(\mathcal{K}(k,i'))) \in \mathsf{WW}_{\mathcal{G}}$; By well-formedness of $\mathcal{G}$ (Eq. (irref-asym-WW)), it cannot be true. Thus we have $i = i'$. Combine step (1), we prove Eq. (4.1).

(3) Given Eq. (5.8) we know that $(t_0, t) \in \mathsf{WW}_{\mathcal{G}}$ for all $t \in \mathcal{G} \wedge t \neq t_0$; and $(\mathtt{W}, k, v_0) \in \mathcal{G}(t_0)$. By the definition of $\mathsf{VerListOf}$, $\mathcal{K}(k,0) = (v_0, t_0, \_)$, which implies Eq. (4.2).

(4) Suppose an index t, t' such that such that $t = \mathsf{WriterOf}(\mathcal{K}(k,i))$ and $t' \in \mathsf{ReadersOf}(\mathcal{K}(k,i))$ for some index i. This means $(t,t') \in \mathsf{WR}_{\mathcal{G}}$. By Eq. (5.4), it follows $(t,t') \notin \mathsf{SO}^{?}$ which implies Eq. (4.3).

(5) Suppose an index t, t' such that such that $t = \mathsf{WriterOf}(\mathcal{K}(k,i))$ and $t' = \mathsf{WriterOf}(\mathcal{K}(k,i'))$ for some indices i, i' with $i < i'$. This means $(t,t') \in \mathsf{WW}_{\mathcal{G}}$. By Eq. (5.9), it follows $(t,t') \notin \mathsf{SO}$ which implies Eq. (4.4). ■

A.3 Proofs for Section 5.2 (Operational Semantics on Abstract Execution)

Proposition A.6 (Well-defined last-write-win resolution policy)**.** *Given a well-formed abstract execution $\mathcal{X}$ and a set of transactions T that includes the initialisation transaction t_0, the function* $\mathsf{AExecSnapshot}(\mathcal{X}, T)$ *is defined.*

Proof. It is straightforward since transaction t_0 initialise all the keys in a well-formed abstract execution. ■

Proposition 5.11 (Well-defined $\mathsf{UpdateAExec}$ function)**.** *Given an abstract execution $\mathcal{X} \in$ AExecs, a set of transactions $T \subseteq \mathcal{X}$ with $t_0 \in T$, a fresh transaction identifier $t \in \mathsf{NextAExecTxID}(\mathcal{K}, cl)$ for some client cl, and a fingerprint $\mathcal{F} \in$ Fp such that*

$$\forall k \in \textsc{Key}.\, \forall v \in \textsc{Value}.\, (\mathtt{R}, k, v) \in \mathcal{F} \Rightarrow \mathsf{MaxVisTrans}(\mathcal{X}, T, k)\,, \tag{A.18}$$

the new abstract execution $\mathsf{UpdateAExec}(\mathcal{X}, T, \mathcal{F}, t)$ *is a well-formed abstract execution.*

Proof. Let new abstract execution $\mathcal{X}' = \mathsf{UpdateAExec}(\mathcal{X}, T, \mathcal{F}, t)$.

(1) [**Case**: **well-formed arbitration order** $\mathsf{AR}_{\mathcal{X}'}$] It suffices to only consider the edges related to the new transaction t. Since the new arbitration order is defined by

$$\mathsf{AR}_{\mathcal{X}'} = \mathsf{AR}_{\mathcal{X}} \cup \{(t', t) \mid t' \in \mathcal{X}'\}$$

and $t_0 \in \mathcal{X}$, it is straightforward for Eqs. (5.12) and (total-AR). Because t is a fresh transaction such that $t \notin \mathcal{X}$, therefore Eq. (irreflexive-AR) holds. By the definition of $\mathsf{AR}_{\mathcal{X}'}$ that contains edges from any transaction t' from the old abstract execution $\mathcal{X}$ to the new transaction t, it follows Eq. (transitive-AR). Last, by definition of $\mathsf{NextAExecTxID}$, the fresh transaction $t = t^n_{cl}$ should be annotated with a bigger number n than transactions for the same client cl in $\mathcal{X}$, that is, $\forall m \in \mathbb{N}.\, \forall t^m_{cl} \in \mathcal{X}.\, n > m$; this means that $\mathsf{SO} \subseteq \mathsf{AR}_{\mathcal{X}'}$ which implies Eq. (5.13).

(2) [**Case**: **well-formed visibility relation** $\mathsf{VIS}_{\mathcal{X}'}$] It suffices to only consider the edges related to the new transaction t. By the definition of $\mathsf{VIS}_{\mathcal{X}'} = \mathsf{VIS}_{\mathcal{X}} \cup \{(t', t) \mid t' \in T\}$ that contains edges from any transaction t' from T to the new transaction t and $t_0 \in T$, it follows Eq. (5.16). The Eq. (5.18) immediately implies Eq. (5.17). Since the fresh transaction identifier are pick by $t \in \mathsf{NextAExecTxID}(\mathcal{X}, cl)$, it is straightforward for Eq. (5.14). Last, by $T \subseteq \mathcal{X}$, it follows Eq. (5.15). ■

Theorem 5.14 (Equal expressibility between declarative and operational semantics on abstract executions)**.** *For any $\mathcal{A} \subseteq$ VisAxioms, the operational semantics capture the same set of abstract executions as direct axiomatic definitions on abstract definitions, that is,* $\bigcup_{\mathtt{P} \in PROGS} [\![\mathtt{P}]\!]_{\mathcal{A}} = \mathsf{ConsisModelAxioms}(\mathcal{A})$.

Proof. This prove is similar to the one in Theorem 4.31. We prove the set closure for both sides respectively.

(1) [**Case**: $\bigcup_{\mathtt{P} \in \mathbf{Progs}} [\![\mathtt{P}]\!]_{\mathcal{A}} \subseteq \mathsf{ConsisModelAxioms}(\mathcal{A})$] By the definition of $\bigcup_{\mathtt{P} \in \mathrm{PROGS}} [\![\mathtt{P}]\!]_{\mathcal{A}}$, It suffices to prove that for every trace π, initial state $(\mathcal{X}_0, \mathcal{E}_0)$ with program $\mathtt{P}$ and final state $(\mathcal{X}, \mathcal{E})$ with program $\mathtt{P}'$,

$$\mathsf{Dom}(\mathtt{P}) \subseteq \mathsf{Dom}(\mathcal{E}_0) \wedge \pi = (\mathcal{X}_0, \mathcal{E}_0), \mathtt{P} \xrightarrow{\iota}{}^{*}_{\mathcal{A}} (\mathcal{X}, \mathcal{E}), \mathtt{P}' \\ \mathcal{X} \in \mathsf{ConsisModelAxioms}(\mathcal{A}). \tag{A.13}$$

We prove Eq. (A.13) by induction on the length of π.

(i) [**Base Case**: $\pi = (\mathcal{X}_0, \mathcal{E}_0), \mathtt{P}$] It is trivial that $\mathtt{AExecSat}(\mathcal{X}_0, \mathcal{A})$. Hence it is easy to see $\mathcal{X}_0 \in \mathsf{ConsisModelAxioms}(\mathcal{A})$.

(ii) [**Inductive Case**: $\pi = \pi' \xrightarrow{\iota}_{\mathcal{A}} (\mathcal{X}, \mathcal{E}), \mathtt{P}'$] Let $\mathsf{Last}(\pi) = (\mathcal{X}', \mathcal{E}'), \mathtt{P}''$. If $\iota = (cl, \bullet)$, then $\mathcal{X} = \mathcal{X}'$; therefore by inductive hypothesis, Eq. (A.13) holds. Consider $\iota = (cl, T, \mathcal{F})$. By the AAtomicTrans rule, $\mathcal{X} = \mathsf{UpdateAExec}(\mathcal{X}', T, \mathcal{F}, t)$ for some $t \in \mathsf{NextAExecTxID}(\mathcal{X}, cl)$. Note that the new abstract execution $\mathcal{X}$ contains all transactions and edges in $\mathcal{X}'$, and thus $\mathcal{X} \simeq_{\mathcal{X}'_{|0}} \mathcal{X}'$. This means that first by the inductive hypothesis, $\mathtt{AExecSat}(\mathcal{X}', \mathcal{A})$, and by the definition of well-formed visibility axioms that every axiom must be local, we only need to consider the new visibility edges. By the last premise of the AAtomicTrans rule, for any $A \in \mathcal{A}$, it must be the case that $A^{-1}(\mathcal{X}')(t) \subseteq T$, and thus $\mathcal{X} \in \mathsf{ConsisModelAxioms}(\mathcal{A})$.

(2) [**Case**: $\bigcup_{\mathtt{P} \in \mathbf{Progs}} [\![\mathtt{P}]\!]_{\mathcal{A}} \supseteq \mathsf{ConsisModelAxioms}(\mathcal{A})$] Assume an abstract execution $\mathcal{X}$ such that $\mathcal{X} \in \mathsf{ConsisModelAxioms}(\mathcal{A})$. Let $(\mathcal{T}, \mathsf{VIS}, \mathsf{AR}) = \mathcal{X}$. We prove a stronger result that there is an abstract execution trace π corresponding to the cut of abstract execution, that

is, for any number i and cut of abstract execution $\mathcal{X}'$

$$\mathcal{X}' = \mathsf{AExectCut}\left(\mathcal{X}, i\right) \Rightarrow \exists \pi.\, \exists \mathtt{P} \in \textsc{Progs}.\, \exists \mathcal{E} \in \textsc{ClientEnvs}.\\ \mathsf{Last}\left(\pi\right) = \left(\mathcal{X}', \mathcal{E}\right), \mathtt{P} \wedge \mathtt{P} = \lambda cl \in \mathsf{Dom}\left(\mathtt{P}\right).\, \mathtt{skip} \tag{A.14}$$

We prove by induction on the number of transactions. The key here is to construct a program $\mathtt{P}$ and this is done in similar way as in Theorem 4.31.

(i) [**Base Case**: $\mathsf{AExectCut}\left(\mathcal{X}, 0\right)$] We know $\mathcal{X}_0 = \mathsf{AExectCut}\left(\mathcal{X}, 0\right)$ by the definition of $\mathcal{X}_0$. Let the client set $C = \{cl \mid \exists n \in \mathbb{N}.\, t_{cl}^n \in \mathcal{X}\}$. We pick a program $\mathtt{P} = \lambda cl \in C.\, \mathtt{skip}$ a client environment $\mathcal{E}_0 = \lambda cl \in C.\, s$ for some stack s, and therefore $\pi = \left(\mathcal{X}_0, \mathcal{E}_0\right), \mathtt{P}$ which implies Eq. (A.14).

(ii) [**Base Case**: $\mathsf{AExectCut}\left(\mathcal{X}, i\right)$] Let $\mathcal{X}'$ and $\mathcal{X}''$ be the $(i-1)^{\text{th}}$ and i^{th} cuts, defined by $\mathcal{X}' = \mathsf{AExectCut}\left(\mathcal{X}, i-1\right)$ and $\mathcal{X}'' = \mathsf{AExectCut}\left(\mathcal{X}, i\right)$. Suppose that there is a trace π' for $\mathsf{AExectCut}\left(\mathcal{X}, i-1\right)$, such that

$$\pi' = \left(\mathcal{X}_0, \mathcal{E}_0\right), \mathtt{P} \;\Rightarrow_{\mathcal{A}}^{*} \left(\mathcal{X}', \mathcal{E}'\right), \mathtt{P}'\\ \wedge\, \mathtt{P}' = \lambda cl \in \mathsf{Dom}\left(\mathtt{P}'\right).\, \mathtt{skip}.$$

Let the transaction $t = \mathcal{X}'' \backslash \mathcal{X}'$, fingerprint $\mathcal{F} = \mathcal{T}_{\mathcal{X}}\left(t\right)$, transaction set $T = \mathsf{VIS}^{-1}\left(t\right)$. By Prop. A.7, it follows that $\mathcal{X}'' = \mathsf{UpdateAExec}\left(\mathcal{X}', T, \mathcal{F}, t\right)$. Let the transactional command $\mathtt{T} = \mathsf{TransFp}\left(\mathcal{F}\right)$ ($\mathsf{TransFp}$ is defined in Theorem 4.31) and the new initial program $\mathtt{P}'' = \mathsf{ExtendProgram}\left(\mathtt{P}, cl, [\,\mathtt{T}\,]\right)$. It is easy to see there exists a new trace π such that

$$\pi = \left(\mathcal{X}_0, \mathcal{E}_0\right), \mathtt{P}'' \;\Rightarrow_{\mathcal{A}}^{*} \left(\mathcal{X}', \mathcal{E}'\right), \mathtt{P}'\left[cl \mapsto [\,\mathtt{T}\,]\right] \xrightarrow{stub}_{\mathcal{A}} \left(\mathcal{X}'', \mathcal{E}''\right), \mathtt{P}^*,$$

for some client environment $\mathcal{E}''$ and finial program $\mathtt{P}^*$ with $\mathtt{P}^* = \lambda cl \in \mathsf{Dom}\left(\mathtt{P}^*\right).\, \mathtt{skip}$.

■

Proposition A.7 (Abstract execution cut to update)**.** *Given an abstract execution $\mathcal{X}$ and the cut $\mathcal{X}' = \mathsf{AExectCut}\left(\mathcal{X}, n\right)$, assume that the transaction t is the next transaction in the arbitration order, that is, $\mathsf{Max}_{\mathsf{AR}_{\mathcal{X}}}\left(\mathcal{X}'\right) \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t$, assume there exists no transaction t' such that $\mathsf{Max}_{\mathsf{AR}_{\mathcal{X}}}\left(\mathcal{X}'\right) \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t' \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t$, and then*

$$\mathsf{AExectCut}\left(\mathcal{X}, n+1\right) = \mathsf{UpdateAExec}\left(\mathcal{X}', \mathsf{VIS}_{\mathcal{X}}^{-1}\left(t\right), t, \mathcal{T}_{\mathcal{X}}\left(t\right)\right).$$

Proof. It is trivial by the definitions of $\mathsf{AExectCut}$ and $\mathsf{UpdateAExec}$. ■

A.4 Proofs for Section 5.3 (Correspondence to Kv-store Semantics)

Proposition A.8 (Well-defined XToD)**.** *Given an abstract execution $\mathcal{X} \in$ AExecTS, the dependency graph $\mathsf{XToD}(\mathcal{X})$ is well-formed.*

Proof. Given the definition of XToD, if the dependency graph $\mathcal{G} = \mathsf{XToD}(\mathcal{X})$ for some abstract execution $\mathcal{X}$, then $\mathsf{WR}_{\mathcal{G}} = \mathsf{WR}_{\mathcal{X}}$, $\mathsf{WW}_{\mathcal{G}} = \mathsf{WW}_{\mathcal{X}}$ and $\mathsf{RW}_{\mathcal{G}} = \mathsf{RW}_{\mathcal{X}}$. Consider all the well-formed conditions for relations $\mathsf{WR}, \mathsf{WW}, \mathsf{RW}$ defined in Def. 5.1.

(1) [**Case**: **write-read dependency** WR] By the definition of $\mathsf{WR}_{\mathcal{X}}$, it is trivial for Eqs. (5.2) and (5.3). Because visibility relation cannot violate session order by Eq. (5.14), it follows Eq. (5.4). Since $\mathsf{MaxVisTrans}$ in the definition of $\mathsf{WR}_{\mathcal{X}}$ returns a unique transaction, it means that Eq. (5.5) holds.

(2) [**Case**: **write-write dependency** WW] By the definition of $\mathsf{WW}_{\mathcal{X}}$, it is trivial for Eqs. (5.7) and (total-WW). Because the initialisation transaction t_0 is $\mathsf{AR}_{\mathcal{X}}$-before any other transactions by Eq. (5.12), this implies Eq. (5.8). Since $\mathsf{AR}_{\mathcal{X}}$ is a total order and $\mathsf{SO} \subseteq \mathsf{AR}_{\mathcal{X}}$, by the definition of $\mathsf{WW}_{\mathcal{X}}$, the write-write dependency cannot violate the SO and thus Eqs. (5.9) and (irref-asym-WW) hold. Last write-write dependency must be transitive, because $\mathsf{AR}_{\mathcal{X}}$ is transitive and $t \xrightarrow{\mathsf{WW}_{\mathcal{X}}(k)} t' \xrightarrow{\mathsf{WW}_{\mathcal{X}}(k)} t''$ implies $t \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t' \xrightarrow{\mathsf{AR}_{\mathcal{X}}} t''$ and therefore $t \xrightarrow{\mathsf{WW}_{\mathcal{X}}(k)} t''$; this implies Eq. (transitive-WW).

(3) [**Case**: **read-write anti-dependency** RW] Consider two transactions t and t' such that $(t, t') \in \mathsf{RW}_{\mathcal{X}}(k)$ for some key k. By definition of $\mathsf{RW}_{\mathcal{X}}$, $(\mathtt{R}, k, v) \in \mathcal{T}_{\mathcal{X}}(t)$ for some value v. There exists a transition t'' such that $t'' = \mathsf{MaxVisTrans}\left(\mathcal{X}, \mathsf{VIS}_{\mathcal{X}}^{-1}(t), k\right)$, and therefore $(t'', t) \in \mathsf{WR}_{\mathcal{X}}(k)$. Again by definition of $\mathsf{RW}_{\mathcal{X}}$, it is must the case that $(t'', t') \in \mathsf{AR}_{\mathcal{X}}$ and therefore $(t'', t') \in \mathsf{WW}_{\mathcal{X}}(k)$.

Now consider three transactions t, t', t'' such that $(t'', t) \in \mathsf{WR}_{\mathcal{X}}(k)$ and $(t'', t') \in \mathsf{WW}_{\mathcal{X}}(k)$ for some key k. First, $(t'', t) \in \mathsf{WR}_{\mathcal{X}}(k)$ implies $t'' = \mathsf{MaxVisTrans}\left(\mathcal{X}, \mathsf{VIS}_{\mathcal{X}}^{-1}(t), k\right)$ and $(t'', t) \in \mathsf{AR}_{\mathcal{X}}$; then because $(t'', t') \in \mathsf{WW}_{\mathcal{X}}(k)$, it means $(t'', t') \in \mathsf{AR}_{\mathcal{X}}(k)$ and thus we have $(t, t') \in \mathsf{RW}_{\mathcal{X}}$ by the definition of $\mathsf{RW}_{\mathcal{X}}$. ■

Theorem 5.17 (Compatibility of $\mathcal{X}$ and $\mathsf{XToK}(\mathcal{X})$)**.** *For any abstract execution $\mathcal{X}$, $\mathcal{X} \sim \mathsf{XToK}(\mathcal{X})$.*

Proof. Let kv-store $\mathcal{K} = \mathsf{XToK}(\mathcal{X})$.

[**Case**: $\mathsf{AExecSnapshot}(\mathcal{X}, T) = \mathsf{Snapshot}(\mathcal{K}, \mathsf{GetView}(\mathcal{X}, T))$ **for all** $T \subseteq \mathcal{X}$ **such that** $t_0 \in T$] By the Prop. A.9, the view $u = \mathsf{GetView}(\mathcal{X}, T)$ is a valid view on $\mathcal{K}$, that is, $u \in \textsc{ViewOn}(\mathcal{K})$. Given that u is a valid view, it is sufficient to prove that for all keys k,

$$\mathsf{AExecSnapshot}(\mathcal{X}, T)(k) = \mathsf{Snapshot}(\mathcal{K}, \mathsf{GetView}(\mathcal{X}, T))(k).$$

Let transaction $t = \mathsf{MaxVisTrans}(\mathcal{X}, T, k)$. Therefore operation $(\mathtt{W}, k, v) \in \mathcal{T}_{\mathcal{X}}(t)$ for some $v \in \textsc{Value}$. By definition of $\mathsf{AExecSnapshot}$, it follows that $\mathsf{AExecSnapshot}(\mathcal{X}, T)(k)$. By definition of $\mathsf{MaxVisTrans}$, any other transaction $t' \in T \setminus \{t\}$ that also wrote the key k with $(\mathtt{W}, k, \_) \in \mathcal{T}_{\mathcal{X}}(t')$, must be $\mathsf{AR}_{\mathcal{X}}$-before t, that is $(t', t) \in \mathsf{AR}_{\mathcal{X}}$; this means that $(t', t) \in \mathsf{WW}_{\mathcal{G}}$ for $\mathcal{G} = \mathsf{XToD}(\mathcal{X})$. Since transactions t, t' both wrote key k, there must exists two indices i, i' such that $t = \mathsf{WriterOf}(\mathcal{K}(k, i))$, $t' = \mathsf{WriterOf}(\mathcal{K}(k, i'))$. Because $\mathcal{K} = \mathsf{DToK}(\mathcal{G})$ and $(t', t) \in \mathsf{WW}_{\mathcal{G}}$, then $i > i'$; therefore by definition of $\mathsf{Snapshot}$, the value matches as $\mathsf{Snapshot}(\mathcal{K}, u)(k) = v$.

[**Case**: $\mathsf{Snapshot}(\mathcal{K}, u) = \mathsf{AExecSnapshot}(\mathcal{X}, \mathsf{VisTrans}(\mathcal{K}, u))$ **for all** $u \subseteq \mathbf{ViewOn}(\mathcal{K})$] Let transaction sets $T = \mathsf{VisTrans}(\mathcal{K}, u)$. It is sufficient to prove that for all keys k,

$$\mathsf{Snapshot}(\mathcal{K}, u)(k) = \mathsf{AExecSnapshot}(\mathcal{X}, \mathsf{VisTrans}(\mathcal{K}, u))(k).$$

Let $v = \mathsf{Snapshot}(\mathcal{K}, u)(k)$ By definition of $\mathsf{VisTrans}$, for versions included in vi for key k, their writer must be included in T, that is

$$\forall i \in u(k) \,.\, \mathsf{WriterOf}(\mathcal{K}(k, i)) \in T.$$

Let t be the transaction that wrote the newest version for key k in u as:

$$t = \mathsf{WriterOf}(\mathcal{K}(k, \mathsf{Max}_{<}(u(k)))) \wedge \forall i \in u(k) \,.\, \mathsf{WriterOf}(\mathcal{K}(k, i)) \xrightarrow{\mathsf{WW}_{\mathcal{K}}^{-1}} t.$$

By definition of $\mathsf{Snapshot}$, the snapshot must include t

$$\mathsf{ValueOf}(\mathcal{K}(k, \mathsf{Max}_{<}(u(k)))) = v,$$

which means $v = \mathsf{AExecSnapshot}(\mathcal{X}, \mathsf{VisTrans}(\mathcal{K}, u))(k)$ by definition of $\mathsf{AExecSnapshot}$. Note that the initial version is always included in the view $0 \in u(k)$, therefore $t_0 \in T$ and the function $\mathsf{AExecSnapshot}(\mathcal{X}, \mathsf{VisTrans}(\mathcal{K}, u))$ must be defined by Prop. A.6. ■

Proposition A.9 (Well-formed views of $\mathsf{GetView}$)**.** *For any abstract execution* $\mathcal{X}$*, and* $T \subseteq \mathcal{X}$*,* $\mathsf{GetView}(\mathcal{X}, T) \in \textit{ViewOn}(\mathsf{XToK}(\mathcal{X}))$.

Proof. Let $u = \mathsf{GetView}(\mathcal{X}, T)$ and $\mathcal{K} = \mathsf{XToK}(\mathcal{X})$. By the definition of $\mathsf{GetView}$, the initial version must be included in the view $0 \in u(k)$ for any key k (Eq. (4.5)). For any index i included in the view for key k such that $i \in u(k)$, it must be the case that $0 \leq i < |\mathcal{K}(k)|$ (Eq. (4.6)).

Now consider two versions $\mathcal{K}(k,i)$ and $\mathcal{K}(k',i')$ such that $i \in u(k)$ and $\mathsf{WriterOf}(\mathcal{K}(k,i)) = \mathsf{WriterOf}(\mathcal{K}(k',i'))$, it must the case that $\mathsf{WriterOf}(\mathcal{K}(k,i)) \in T$ thus $i' \in u(k')$ (Eq. (4.7)). ■

Theorem 5.19 (Well-formed abstract executions of XToTrace)**.** *Given an* $\mathtt{ET}_\top$*-trace* τ*, an abstract execution* $\mathcal{X}$ *such that* $\mathcal{X} \in \mathsf{TraceToX}(\tau)$*, is a well-formed abstract execution and* $\mathsf{Last}(\tau)_{|0} = \mathsf{XToK}(\mathcal{X})$.

Proof. We prove by induction on the length of the trace τ.

(1) [**Base Case**: $\tau = (\mathcal{K}_0, \mathcal{U}_0)$] By definition of TraceToX, the only abstract execution $\{\mathcal{X}\} = \mathsf{TraceToX}(\tau)$ is defined by $\mathcal{X} = (\{t_0 \mapsto \{(k, v_0) \mid k \in \textsc{Key} \wedge v_0 \in \mathsf{InitialValue}(k)\}\}, \emptyset, \emptyset)$; it is trivially a well-formed abstract execution and $\mathcal{K}_0 = \mathsf{XToK}(\mathcal{X})$.

(2) [**Inductive Case**: $\tau = \tau' \xrightarrow{\iota}_\top (\mathcal{K}, \mathcal{U})$] Consider the label ι.

(i) [**Case**: $\iota = (cl, u)$] By definition of TraceToX, $\mathsf{TraceToX}(\tau) = \mathsf{TraceToX}(\tau')$; by inductive hypothesis, every abstract execution in $\mathsf{TraceToX}(\tau)$ is well-formed.

(ii) [**Case**: $\iota = (cl, \mathcal{F})$] Let configuration $(\mathcal{K}', \mathcal{U}') = \mathsf{Last}(\tau')$ and view $u = \mathcal{U}'(cl)$. By inductive hypothesis, suppose a well-formed abstract execution $(\mathcal{T}', \mathsf{VIS}', \mathsf{AR}') \in \mathsf{TraceToX}(\tau')$. Let a set of read only transactions T_{rd} such that

$$T_{\mathrm{rd}} \subseteq \{t' \mid \forall l.\, \forall k \in \textsc{Key}.\, \forall v \in \textsc{Value}.\, (l, k, v) \in \mathcal{T}'(t') \Rightarrow l = \mathtt{R}\}.$$

Let the set of visible transactions $T = \mathsf{GetView}(\mathcal{K}', u') \cup T_{\mathrm{rd}}$. Assume new transaction $t' = \mathcal{K} \backslash \mathcal{K}'$. By definitions of TraceToX and UpdateAExec, the new abstract execution is defined by

$$\mathcal{X} = \left(\mathcal{T} \cup \{t' \mapsto \mathcal{F}\}, \mathsf{VIS}' \cup \{(t', t) \mid t' \in T\}, \mathsf{AR}' \cup \{(t', t) \mid t' \in \mathcal{X}\}\right).$$

Let $\mathsf{VIS} = \mathsf{VIS}' \cup \{(t', t) \mid t' \in T\}$ and $\mathsf{AR} = \mathsf{AR}' \cup \{(t', t) \mid t' \in \mathcal{X}\}$. By inductive hypothesis, we only need to consider the arbitration and visibility edges related to the new transaction t. It is trivial that the new arbitrary relation AR satisfies Eqs. (5.12) and (total-AR) to (transitive-AR); because the new transition is picked by $t \in \mathsf{NextTxID}(\mathcal{K}', cl)$, Eq. (5.13) holds. Consider the new visibility relation $\mathsf{VIS}' \cup \{(t', t) \mid t' \in T\}$. By the definition of GetView, the initialisation transaction $t_0 \in T$, and thus $(t_0, t) \in \mathsf{VIS}$ which implies *Eq.* (5.16). Consider a read operation $(\mathtt{R}, k, v) \in \mathcal{F}$ for some key k and v. By the $\mathtt{ET}_\top$-trace and UpdateKV function, it is known that $\mathcal{K}'(k, \mathsf{Max}_<(\mathcal{K}, u)) = v$; therefore by the definition of GetView, the writer $\mathsf{WriterOf}(\mathcal{K}'(k, \mathsf{Max}_<(\mathcal{K}, u))) = \mathsf{MaxVisTrans}\left((\mathcal{T}', \mathsf{VIS}', \mathsf{AR}'), T, k\right)$, which implies Eq. (5.17). Because the new transition is picked by $t \in \mathsf{NextTxID}(\mathcal{K}', cl)$, Eq. (5.14) holds. Last, it is trivial that $\mathsf{VIS} \subseteq \mathsf{AR}$.

Now we prove $\mathcal{K} = \mathsf{XToK}(\mathcal{X})$ for the new abstract execution $\mathcal{X}$. Note that $\mathcal{X} = \mathsf{UpdateAExec}(\mathcal{X}, T, \mathcal{F}, t)$ for a fresh transaction identifier $tin\mathsf{NextAExecTxID}(\mathcal{X}, cl)$, and $\mathcal{K} = \mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t)$ for the view $u = \mathsf{GetView}(T, \mathcal{K})$. By Prop. A.10, $\mathcal{K} = \mathsf{XToK}(\mathcal{X})$. ■

Proposition A.10 (Update of abstract execution matching update of kv-store)**.** *Assume a kv-store $\mathcal{K}$ and an abstract execution $\mathcal{X}$ such that $\mathcal{K} = \mathsf{XToK}(\mathcal{X})$. Let $u = \mathsf{GetView}(\mathcal{K}, T)$ for some view u and transaction set $T \subseteq \mathcal{X}$ with $t_0 \in T$. Given the new abstract execution $\mathcal{X}' = \mathsf{UpdateAExec}(\mathcal{X}, T, \mathcal{F}, t)$ for some fingerprint $\mathcal{F}$ and fresh transaction $t \in \mathsf{NextAExecTxID}(\mathcal{X}, cl)$, then $\mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t) = \mathsf{XToK}(\mathcal{X}')$.*

Proof. Let $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t)$. Note that the fresh transaction $t \in \mathsf{NextAExecTxID}(\mathcal{X}, cl)$ then $t \in \mathsf{NextTxID}(\mathcal{K}, cl)$. By Theorem 5.4 and Def. 5.15, it suffice to prove that $\mathsf{KToD}(\mathcal{K}') = \mathsf{XToD}(\mathcal{X}')$. By the definition of KToD and XToD in Defs. 5.2 and 5.15 respectively, it is sufficient to prove that:

$$\mathsf{KToD}(\mathcal{K}) = \mathsf{XToD}(\mathcal{X}) \Rightarrow \mathsf{KToD}(\mathcal{K}')_{|0} = \mathsf{KToD}(\mathcal{K})_{|0}\{t \mapsto \mathcal{F}\} = \mathsf{XToD}(\mathcal{X}')_{|0} \\ \wedge \mathsf{WR}_{\mathcal{K}'} = \mathsf{WR}_{\mathcal{X}'} \wedge \mathsf{WW}_{\mathcal{K}'} = \mathsf{WW}_{\mathcal{X}'}. \quad \text{(A.15)}$$

Recall that in dependency graphs, RW can be derived from WW and WW. Consider the three conjunctions.

(1) [**Case**: **new transaction t and its fingerprint $\mathcal{F}$**] By definition of $\mathsf{UpdateAExec}$, it is known that, for any operation $o \in \mathcal{F}$ key k and value v,

$$(o = (\mathtt{R}, k, v) \Leftrightarrow (\mathtt{R}, k, v) \in \mathcal{X}'(t)) \wedge (o = (\mathtt{W}, k, v) \Leftrightarrow (\mathtt{W}, k, v) \in \mathcal{X}'(t)).$$

then by definition of XToD, we know that

$$\mathsf{XToD}(\mathcal{X})_{|0}\{t \mapsto \mathcal{F}\} = \mathsf{XToD}(\mathcal{X}')_{|0}.$$

Now consider the another side. By the definition of $\mathsf{UpdateKV}$, for any operation $o \in \mathcal{F}$, we have the following result for some key k and v:

$$(o = (\mathtt{R}, k, v) \Leftrightarrow \exists i.\, t \in \mathsf{ReadersOf}(\mathcal{K}'(k, i))) \\ \wedge (o = (\mathtt{W}, k, v) \Leftrightarrow t = \mathsf{WriterOf}(\mathcal{K}'(k, |\mathcal{K}'(k)| - 1))).$$

Therefore by definition of KToD,

$$(\exists i.\, t \in \mathsf{ReadersOf}(\mathcal{K}'(k, i)) \Leftrightarrow (\mathtt{R}, k, v) \in \mathsf{KToD}(\mathcal{K}')) \\ \wedge (t = \mathsf{WriterOf}(\mathcal{K}'(k, |\mathcal{K}'(k)| - 1)) \Leftrightarrow (\mathtt{W}, k, v) \in \mathsf{KToD}(\mathcal{K}')),$$

which means that $\mathsf{KToD}\left(\mathcal{K}'\right)_{|0} = \mathsf{KToD}\left(\mathcal{K}\right)_{|0}\left\{t \mapsto \mathcal{F}\right\}$. Recall that $\mathsf{KToD}\left(\mathcal{K}\right) = \mathsf{XToD}\left(\mathcal{X}\right)$; this means $\mathsf{KToD}\left(\mathcal{K}'\right) = \mathsf{XToD}\left(\mathcal{X}'\right)$.

(2) [**Case**: **write-read dependency** WR] Because $\mathsf{KToD}\left(\mathcal{K}\right) = \mathsf{XToD}\left(\mathcal{X}\right)$, we only need to consider WR-edges related to the new transaction t. Consider any key k such that $(\mathtt{R}, k, v) \in \mathcal{F}$ for some value v. By definition of UpdateAExec, transaction t must read from a transaction $t' \in T$ who wrote the key k and is the latest transaction visible by t, and therefore by definition of XToD,

$$t' = \mathsf{MaxVisTrans}\left(\mathcal{X}, T, k\right) \Leftrightarrow (t', t) \in \mathsf{WR}_{\mathcal{X}'}.$$

By definition of the view $u = \mathsf{GetView}\left(\mathcal{X}, T\right)$, the latest version of k must be written by t, that is,

$$t' = \mathsf{WriterOf}\left(\mathcal{K}\left(k, \mathsf{Max}_{<}\left(u\left(k\right)\right)\right)\right).$$

By the definition of UpdateKV, the transaction t must read the version $\mathcal{K}\left(k, \mathsf{Max}_{<}\left(u\left(k\right)\right)\right)$, and then by definition of KToD,

$$(t' = \mathsf{WriterOf}\left(\mathcal{K}'(k, \mathsf{Max}_{<}\left(u\left(k\right)\right))\right) \wedge t \in \mathsf{ReadersOf}\left(\mathcal{K}'(k, \mathsf{Max}_{<}\left(u\left(k\right)\right))\right)) \Leftrightarrow (t', t) \in \mathsf{WR}_{\mathcal{K}'}.$$

Given above, we know

$$(t', t) \in \mathsf{WR}_{\mathcal{X}'} \Leftrightarrow (t', t) \in \mathsf{WR}_{\mathcal{K}'}$$

which means that for any new write-read dependency $\mathsf{WR}_{\mathcal{X}'}$, the same edge exists in $\mathsf{WR}_{\mathcal{K}'}$ and vice versa. Since $\mathsf{WR}_{\mathcal{X}} = \mathsf{WR}_{\mathcal{K}}$, we know $\mathsf{WR}_{\mathcal{X}'} = \mathsf{WR}_{\mathcal{K}'}$.

(3) [**Case**: **write-write dependency** WW] Because $\mathsf{KToD}\left(\mathcal{K}\right) = \mathsf{XToD}\left(\mathcal{X}\right)$, we only need to consider WW-edges related to the new transaction t. Consider any key k such that $(\mathtt{W}, k, v) \in \mathcal{F}$ for some value v. By the definition of *UpdateAExec*, the new transaction wrote the key k, and then by definition of XToD,

$$(\mathtt{W}, k, v) \in \mathcal{X}'(t) \Leftrightarrow \forall t' \in \mathcal{X}.\, \forall v' \in \textsc{Value}.\, ((\mathtt{W}, k, v') \in \mathcal{T}_{\mathcal{X}}\left(t'\right) \Rightarrow (t', t) \in \mathsf{AR}_{\mathcal{X}'}).$$

Since $\mathcal{K} = \mathsf{XToK}\left(\mathcal{X}\right)$, if a transaction wrote the key k in $\mathcal{X}$, it must wrote a version of the key k in $\mathcal{K}$, that is,

$$\forall t' \in \mathcal{X}.\, \forall v' \in \textsc{Value}.\, (\mathtt{W}, k, v') \in \mathcal{T}_{\mathcal{X}}\left(t'\right) \Leftrightarrow \exists i \in \mathbb{N}.\, t' = \mathsf{WriterOf}\left(\mathcal{K}\left(k, i\right)\right).$$

By the definition of UpdateKV, the transaction t wrote the latest version of k in the new kv-store $\mathcal{K}$, and by the definition of KToD,

$$\begin{aligned} &t = \mathsf{WriterOf}\left(\mathcal{K}'(k, |\mathcal{K}'(k)| - 1)\right) \Leftrightarrow \\ &\quad \forall i \in \mathbb{N}.\, (0 \leq i < |\mathcal{K}'(k)| - 1 \Rightarrow \mathsf{WriterOf}\left(\mathcal{K}'(k, i)\right) \xrightarrow{\mathsf{WW}_{\mathcal{K}}} t). \end{aligned}$$

Note that $|\mathcal{K}'(k)| - 1 = |\mathcal{K}(k)|$. Therefore for any new write-write dependency $\mathsf{WW}_{\mathcal{X}'}$, the same edge exists in $\mathsf{WW}_{\mathcal{K}'}$ and vice versa. Since $\mathsf{WW}_{\mathcal{X}} = \mathsf{WW}_{\mathcal{K}}$, we know $\mathsf{WW}_{\mathcal{X}'} = \mathsf{WW}_{\mathcal{K}'}$. ■

Proposition A.11 (Well-defined ApproxView)**.** *Given an abstract execution $\mathcal{X}$, let $\mathcal{X}' = \mathsf{AExectCut}(\mathcal{X}, i)$ and $u = \mathsf{ApproxView}(\mathcal{X}, i, t)$ for some number i and transaction t. If t is the last transaction with respect to arbitration order, $t = \mathsf{Max}_{\mathsf{AR}_{\mathcal{X}'}}(\mathcal{X}')$, then the view $u \in \textsc{ViewOn}(\mathsf{XToK}(\mathcal{X}'))$.*

Proof. It is trivial from the definition of ApproxView. ■

Theorem 5.21 (Abstract executions to well-formed $\mathtt{ET}_\top$-traces)**.** *Given an abstract execution $\mathcal{X}$, any trace $\tau \in \mathsf{XToTrace}(\mathcal{X})$ is a valid $\mathtt{ET}_\top$-trace and the final kv-store $\mathcal{K}$ such that $(\mathcal{K}, \_) = \mathsf{Last}(\tau)$ satisfies that $\mathcal{K} = \mathsf{XToK}(\mathcal{X})$.*

Proof. Given the definition of XToTrace, we prove a stronger result that, for any number i, kv-store $\mathcal{K}$, view environment $\mathcal{U}$ and trace τ,

$$\tau = \mathsf{XToTraceN}(\mathcal{X}, i) \wedge (\mathcal{K}, \mathcal{U}) = \mathsf{Last}(\tau) \wedge \mathcal{K} = \mathsf{XToK}(\mathsf{AExectCut}(\mathcal{X}, i)) \tag{A.16}$$

We prove by induction on the number i.

(1) [**Base Case**: $i = 0$] Any trace $\tau \in \mathsf{XToTraceN}(\mathcal{X}, 0)$ is of the form $(\mathcal{K}_0, \mathcal{U}_0)$. By definition of AExectCut, it follows $\mathcal{K}_0 = \mathsf{XToK}(\mathsf{AExectCut}(\mathcal{X}, 0))$.

(2) [**Base Case**: $i > 0$] Suppose that Eq. (A.16) holds for $i - 1$ and consider i. Let $\mathcal{X}' = \mathsf{AExectCut}(\mathcal{X}, i-1)$ and $\mathcal{X}'' = \mathsf{AExectCut}(\mathcal{X}, i)$. Assume the new transaction $t^n_{cl} = \mathcal{X}'' \setminus \mathcal{X}'$ and its fingerprint $\mathcal{F} = \mathcal{T}_{\mathcal{X}}(t^n_{cl})$. By inductive hypothesis, assume a valid $\mathtt{ET}_\top$-trace $\tau \in \mathsf{XToTraceN}(\mathcal{X}, i)$ and its last configuration $(\mathcal{K}, \mathcal{U}) = \mathsf{Last}(\tau)$. Let the view $u = \mathsf{GetView}\left(\mathcal{X}, \mathsf{VIS}^{-1}(t^n_{cl})\right)$, the new kv-store $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t^n_{cl})$, and the new view $u' = \mathsf{ApproxView}(\mathcal{X}, i, t^n_{cl})$. Therefore the new trace τ' is of the form

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

We now prove the two new steps, (cl, u) and $(cl, \mathcal{F})$, satisfy the definition of $\mathtt{ET}_\top$-trace in Def. 4.26.

(i) [**Case**: (cl, u)] By the definition of XToTraceN, there must exist a transaction t^l_{cl} for the client cl such that

$$\begin{aligned}\tau = \cdots \Rightarrow_{\mathtt{ET}} (\mathcal{K}^*, \mathcal{U}^*) \\ \xrightarrow{(cl,\mathcal{F}^*)}_{\mathtt{ET}} \left(\mathsf{UpdateKV}\left(\mathcal{K}^*, \mathcal{U}^*(cl), \mathcal{F}^*, t^l_{cl}\right), \mathcal{U}^*[cl \mapsto u^*]\right) \Rightarrow_{\mathtt{ET}} \tau^{**} \\ \wedge \forall j.\, 0 \le j < |\tau^{**}| \wedge (\tau'_{|j})_{|1}(cl) = u^*,\end{aligned}$$

which means that t_{cl}^m is the last transaction for cl in the trace τ. Again, by the definitions of $\mathsf{XToTraceN}$ and $\mathsf{ApproxView}$, for the view u, there exists a set of transactions T such that,

$$u^* = \mathsf{GetView}\left(\mathcal{X}, T\right) \wedge T \subseteq \mathsf{VIS}_{\mathcal{X}}^{-1}\left(t_{cl}^n\right) \cap \left(\mathsf{AR}_{\mathcal{X}}^{-1}\right)^?\left(t_{cl}^j\right).$$

Note that u^* must be a valid view because of Prop. A.11. Last, by definition of $\mathsf{GetView}$,

$$u^* = \mathsf{GetView}\left(\mathcal{X}, T\right) \subseteq \mathsf{GetView}\left(\mathcal{X}, \mathsf{VIS}_{\mathcal{X}}^{-1}\left(t_{cl}^n\right)\right) = u$$

(ii) [**Case**: $(cl, \mathcal{F})$ **and** $\mathcal{K} = \mathsf{XToK}\left(\mathsf{AExectCut}\left(\mathcal{X}, i\right)\right)$] We prove that the new kv-store $\mathcal{K}'$ is well-formed, $\mathcal{K} = \mathsf{XToK}\left(\mathcal{X}''\right)$ and $(\mathcal{K}, u) \xrightarrow{\mathcal{F}}_{\top} (\mathcal{K}', u')$. The first and third sub-goals imply that $(cl, \mathcal{F})$ is a valid $\mathtt{ET}_{\top}$ step.

(a) [**Case**: **Well-formed** $\mathcal{K}'$] Consider view u defined by $u = \mathsf{GetView}\left(\mathcal{X}, \mathsf{VIS}_{\mathcal{X}}^{-1}\left(t_{cl}^n\right)\right)$. Since $t_{cl}^n = \mathsf{Max}_{\mathsf{AR}_{\mathcal{X}''}}\left(\mathcal{X}''\right)$ and $\mathcal{X}''$ is well-formed, then $\mathsf{VIS}_{\mathcal{X}}^{-1}\left(t_{cl}^n\right) \subseteq \mathcal{X}'$, which means that all visible transactions must exist in $\mathcal{X}' = \mathsf{AExectCut}\left(\mathcal{X}, i-1\right)$. By inductive hypothesis, $\mathcal{K} = \mathsf{XToK}\left(\mathcal{X}'\right)$ and therefore $u \in \textsc{ViewOn}\left(u\right)$. It is easy to see that $t_{cl}^n \notin \mathcal{K}$ and $\mathcal{F} \in \textsc{Fp}$; thus by Theorem 4.21, the new kv-store $\mathcal{K}'$ is well-formed.

(b) [**Case**: $\mathcal{K}' = \mathsf{XToK}\left(\mathcal{X}''\right)$] Let $T = \mathsf{VIS}_{\mathcal{X}}^{-1}\left(t_{cl}^n\right)$. Previously, we known that $u = \mathsf{GetView}\left(\mathcal{X}, T\right)$ and $T \subseteq \mathcal{X}'$; therefore $u = \mathsf{GetView}\left(\mathcal{X}', T\right)$. By Prop. A.7, $\mathcal{X}'' = \mathsf{UpdateAExec}\left(\mathcal{X}', T, \mathcal{F}, t_{cl}^n\right)$. By inductive hypothesis, $\mathcal{K} = \mathsf{XToK}\left(\mathcal{X}'\right)$. Then by Prop. A.10, $\mathcal{K}' = \mathsf{XToK}\left(\mathcal{X}''\right)$.

(c) [**Case**: $(\mathcal{K}, u) \xrightarrow{\mathcal{F}}_{\top} (\mathcal{K}', u')$] Previously, we proved $u \in \textsc{ViewOn}\left(\mathcal{K}\right)$ and $\mathcal{K}' = \mathsf{XToK}\left(\mathcal{X}''\right)$. Then by Prop. A.11, we know that $u' \in \textsc{ViewOn}\left(\mathcal{K}'\right)$. Because $\mathtt{CanCommit}_{\top}$ and $\mathtt{ViewShift}_{\top}$ are simply true, it remains to prove that every read operation $(\mathtt{R}, k, v)$ agrees with the view u. Note that by Prop. A.7, $\mathcal{X}'' = \mathsf{UpdateAExec}\left(\mathcal{X}', \mathsf{VIS}_{\mathcal{X}}^{-1}\left(t_{cl}^n\right), \mathcal{F}, t_{cl}^n\right)$. Given the definition of $\mathsf{UpdateAExec}$,

$$\begin{aligned}&(\mathtt{R}, k, v) \in \mathcal{X}''(t_{cl}^n) \Leftrightarrow \\ &\quad \exists t \in \mathcal{X}'.\, t = \mathsf{MaxVisTrans}\left(\mathcal{X}', \mathsf{VIS}_{\mathcal{X}}^{-1}\left(t_{cl}^n\right), k\right) \wedge (\mathtt{W}, k, v) \in \mathcal{X}'(t).\end{aligned}$$

Because $u = \mathsf{GetView}\left(\mathcal{X}, \mathsf{VIS}_{\mathcal{X}}^{-1}\left(t_{cl}^n\right)\right)$, then

$$(\mathtt{R}, k, v) \in \mathcal{X}''(t_{cl}^n) \Leftrightarrow \mathsf{ValueOf}\left(\mathcal{K}\left(k, \mathsf{Max}_{<}\left(u\left(k\right)\right)\right)\right) = v.$$

Given above we proved $(\mathcal{K}, u) \xrightarrow{\mathcal{F}}_{\top} (\mathcal{K}', u')$. ■

A.5 Proofs for Section 5.4 (Soundness and Completeness Constructors)

Theorem 5.24 (Soundness of execution tests)**.** *Given an execution test* ET *is sound with respect to a set of visibility axioms* $\mathcal{A}$*, then*

$$\bigcup_{\mathtt{P}\in \mathit{PROGS}} [\![\mathtt{P}]\!]_{\mathtt{ET}} \subseteq \{\mathsf{XToK}\,(\mathcal{X}) \mid \mathcal{X} \in \mathsf{ConsisModelAxioms}\,(\mathcal{A})\}\,.$$

Proof. By Theorem 4.31 stating that $\bigcup_{\mathtt{P}\in \mathrm{PROGS}} = \mathsf{ConsisModel}\,(\mathtt{ET})$, it suffices to prove the following result:

$$\mathsf{ConsisModel}\,(\mathtt{ET}) \subseteq \{\mathsf{XToK}\,(\mathcal{X}) \mid \mathcal{X} \in \mathsf{ConsisModelAxioms}\,(\mathcal{A})\}\,.$$

Let I be the invariant that gives rise of soundness constructor. We prove a stronger result that: for any ET-trace τ, there exists an abstract execution $\mathcal{X}$ that satisfies $\mathcal{A}$ and preserves I, that is,

$$\begin{aligned}\tau = (\mathcal{K}_0,\mathcal{U}_0) \Rightarrow^*_{\mathtt{ET}} (\mathcal{K},\mathcal{U}) \wedge \mathcal{K} = \mathsf{XToK}\,(\mathcal{X}) \wedge \mathcal{X} \in \mathsf{ConsisModelAxioms}\,(\mathcal{A}) \\ \wedge\, \forall cl \in \mathsf{Dom}\,(\mathcal{U})\,.\, I\,(\mathcal{X}, cl) \subseteq \mathsf{VisTrans}\,(\mathcal{K},\mathcal{U}\,(cl))\end{aligned} \tag{A.17}$$

First, the trace τ must be a $\mathtt{ET}_\top$-trace, and by Theorem 5.19 for any $\mathcal{X}' \in \mathsf{TraceToX}\,(\tau)$. Hence, the final kv-store $\mathcal{K}$ satisfies that $\mathcal{K} = \mathsf{XToK}\,(\mathcal{X}')$. We now show that we can always find an abstract execution $\mathcal{X} \in \mathsf{TraceToX}\,(\tau)$ such that $\mathtt{AExecSat}\,(\mathcal{X},\mathcal{A})$ and $I\,(\mathcal{X}, cl) \subseteq T_{cl}$ for any client cl and set of transactions $T_{cl} = \mathsf{VisTrans}\,(\mathcal{K},\mathcal{U}\,(cl)) \cup T_{\mathrm{rd}}$ and read-only transactions T_{rd} in $\mathcal{X}$. We prove it by induction on the length of τ. Note that by Theorem 4.30, it is safe to assume that τ is a normal trace, that is, $\mathtt{NormalisedTrace}\,(\tau)$.

(1) [**Base Case**: $\tau = (\mathcal{K}_0,\mathcal{U}_0)$] By definition of $\mathsf{TraceToX}$, the initial abstract execution $\mathcal{X}_0$ satisfies that $\mathcal{K}_0 = \mathsf{XToK}\,(\mathcal{X}_0)$ and $\mathcal{X}_0 \in \mathsf{ConsisModelAxioms}\,(\mathcal{A})$. For any client cl, by the definition $\mathcal{U}_0$, we know that $\mathsf{VisTrans}\,(\mathcal{K},\mathcal{U}\,(cl)) = \{t \mid t \in \mathcal{K}_0\}$, and by the well-formed condition of I, therefore $I\,(\mathcal{X}, cl) \subseteq \mathsf{VisTrans}\,(\mathcal{K},\mathcal{U}\,(cl))$.

(2) [**Inductive Case**: $\tau = \tau' \xrightarrow{(cl,u)}_{\mathtt{ET}} (\mathcal{K},\mathcal{U}) \xrightarrow{(cl,\mathcal{F})}_{\mathtt{ET}} (\mathcal{K}'',\mathcal{U}'')$] Let $(\mathcal{K}',\mathcal{U}')$ be the last configuration, defined by $(\mathcal{K}',\mathcal{U}') = \mathsf{Last}\,(\tau')$. By inductive hypothesis, there exists $\mathcal{X}' \in \mathsf{ConsisModelAxioms}\,(\mathcal{A})$ such that $\mathcal{K}' \in \mathsf{XToK}\,(\mathcal{X}')$ and $I\,(\mathcal{X}, cl') \subseteq \mathsf{VisTrans}\,(\mathcal{K}',\mathcal{U}\,(cl')) \cup T_{\mathrm{rd}}$ for any client $cl' \in \mathsf{Dom}\,(\mathcal{U})$ and some read only transaction set T_{rd}. Now consider the next two steps, (cl, u) and $(cl, \mathcal{F})$, for respectively.

(i) [**Case**: (cl, u)] By the definition of view-shift, $\mathcal{K} = \mathcal{K}'$ and thus $\mathcal{K} = \mathsf{XToK}\,(\mathcal{X})$. Again, by the definition of view-shift $\mathcal{U} = \mathcal{U}'\,[cl \mapsto u]$ and $\mathcal{U}\,(cl) \sqsubseteq u$. Since

$I(\mathcal{X}, cl) \subseteq \mathsf{VisTrans}(\mathcal{K}, \mathcal{U}(cl)) \cup T_{\mathrm{rd}}$ for some read-only transaction set T_{rd}, then $I(\mathcal{X}, cl) \subseteq \mathsf{VisTrans}(\mathcal{K}, u) \cup T_{\mathrm{rd}}$. Last, the invariant for client cl' other than cl is trivially preserved.

(ii) [**Case**: $(cl, \mathcal{F})$] Let $u'' = \mathcal{U}''(cl)$ be the new view and T be a set of transactions defined by $T = \mathsf{VisTrans}(\mathcal{K}, u)$. By the definition of $(cl, \mathcal{F})$, the new kv-store $\mathcal{K}'' = \mathsf{UpdateKV}(\mathcal{K}, u, \mathcal{F}, t)$ for some $t \in \mathsf{NextTxID}(\mathcal{K}, cl)$, and this update satisfies $\mathtt{ET}$, that is, $(\mathcal{K}, u) \xrightarrow{\mathcal{F}}_{\mathtt{ET}} (\mathcal{K}'', u'')$. Since $\mathtt{ET}$ is sound with respect to $\mathcal{A}$, by Def. 5.23, there must exists a set of read-only transactions T_{rd} and a new abstract execution $\mathcal{X}' = \mathsf{UpdateAExec}(\mathcal{X}, T \cup T_{\mathrm{rd}}, \mathcal{F}, t)$ such that Eqs. (5.24) and (5.25) hold. Note that τ is also a $\mathtt{ET}_{\top}$-trace and $\mathcal{X}' in \mathsf{TraceToX}(\tau)$; then by Theorem 5.19, we know that $\mathcal{X}'$ is a well-formed abstract execution and $\mathcal{K}'' = \mathsf{XToK}(\mathcal{X}')$.

Now we prove that $\mathcal{X}' \in \mathsf{ConsisModelAxioms}(\mathcal{A})$. Fix a visibility axiom $A \in \mathcal{A}$. Assume two transactions t', t'' such that $(t', t'') \in A(\mathcal{X}')$. Note that $t' \neq t$ since $\mathsf{VIS}_{\mathcal{X}'} \subseteq \mathsf{AR}_{\mathcal{X}'}$. (a) [**Case**: $t'' \neq t$] For this case, we know $t', t'' \in \mathcal{X}$. By inductive hypothesis that $\mathcal{X} \in \mathsf{ConsisModelAxioms}(\mathcal{A})$ and the well-formed condition for $\mathcal{A}$, it follows that $(t', t'') \in \mathsf{VIS}_{\mathcal{X}'}$. (b) [**Case**: $t'' = t$] By Eq. (5.24) and definition of $\mathsf{UpdateAExec}$, then $(t', t'') \in \mathsf{VIS}_{\mathcal{X}'}$.

Last, we prove that $I(\mathcal{X}', cl') \subseteq \mathsf{VisTrans}(\mathcal{K}'', \mathcal{U}''(cl'))$ for any client $cl' \in \mathsf{Dom}(\mathcal{U}'')$. If $cl' \neq cl$, by inductive hypothesis, the invariant holds. Otherwise, $cl' = cl$ and by Eq. (5.25) the invariant holds. ■

Theorem 5.26 (Completeness of execution tests)**.** *Given an execution test* $\mathtt{ET}$ *that is complete with respect to a set of visibility axioms* $\mathcal{A}$*, then* $\{\mathsf{XToK}(\mathcal{X}) \mid \mathcal{X} \in \mathsf{ConsisModelAxioms}(\mathcal{A})\} \subseteq \bigcup_{\mathtt{P} \in PROGS} [\![\mathtt{P}]\!]_{\mathcal{A}}$.

Proof. By Theorem 4.31 stating that $\bigcup_{\mathtt{P} \in \mathrm{PROGS}} [\![\mathtt{P}]\!]_{\mathcal{A}} = \mathsf{ConsisModel}(\mathtt{ET})$, it suffices to prove the following result

$$\{\mathsf{XToK}(\mathcal{X}) \mid \mathcal{X} \in \mathsf{ConsisModelAxioms}(\mathcal{A})\} \subseteq \mathsf{ConsisModel}(\mathtt{ET}). \tag{A.18}$$

For Eq. (A.18), it is sufficient to prove that, for any abstract execution $\mathcal{X}$ that satisfies $\mathcal{A}$, that is $\mathcal{X} \in \mathsf{ConsisModelAxioms}(\mathcal{A})$, there exists an $\mathtt{ET}$-trace τ such that $\mathsf{Last}(\tau)_{|0} = \mathsf{XToK}(\mathcal{X})$. We prove a stronger result (an invariant) as the following: for any cut of $\mathcal{X}$, namely $\mathcal{X}' = \mathsf{AExectCut}(\mathcal{X}, i)$, there exists an $\mathtt{ET}$-trace τ such that, for any kv-store $\mathcal{K}$, view environment $\mathcal{U}$ and a set of clients CL,

$$CL = \{cl \mid \exists t^n_{cl}.\, t^n_{cl} \in \mathcal{X}\} \wedge (\mathcal{K}, \mathcal{U}) = \mathsf{Last}(\tau) \Rightarrow \mathcal{K} = \mathsf{XToK}(\mathcal{X}') \wedge \left(\begin{array}{l} \forall cl \in CL.\, \forall t \in \mathcal{X}'.\, t = \mathsf{Max}_{\mathsf{AR}_{\mathcal{X}}}(\{t' \mid t \in \mathcal{X}' \wedge t' = t_{\overline{cl}}\}) \Rightarrow \exists T' \subseteq (\mathsf{AR}^{-1}_{\mathcal{X}})^{?}(t). \\ \quad \left(\exists t''.\, t'' = \mathsf{Min}_{\mathsf{SO}}(\{t' \mid t' \in \mathcal{X} \wedge (t, t') \in \mathsf{SO}\}) \Rightarrow T' \subseteq \mathsf{VIS}^{-1}_{\mathcal{X}}(t'')\right) \\ \qquad\qquad \wedge \mathcal{U}(cl) = \mathsf{GetView}(\mathcal{X}, T') \end{array} \right). \tag{A.19}$$

Since $\mathcal{X} \in \mathsf{ConsisModelAxioms}(\mathcal{A})$, then $\mathcal{X} \in \mathsf{ConsisModelAxioms}(\mathcal{A}_\top)$. By Def. 5.20 and Theorem 5.21, any trace included in $\mathsf{XToTrace}(\mathcal{X})$ must be a valid $\mathtt{ET}_\top$-trace and compatible with $\mathcal{X}$. We now prove that τ is a $\mathtt{ET}$-trace and preserves the invariant in Eq. (A.19) by induction one the cut i.

(1) [**Base Case**: $\mathcal{X}' = \mathsf{AExectCut}(\mathcal{X}, 0)$] By definition of $\mathsf{AExectCut}$, we know that $\mathcal{X}' = \mathcal{X}_0$. We construct an $\mathtt{ET}$-trace τ such that $\tau = (\mathcal{K}_0, \mathcal{U}_0)$ and $\mathcal{U}_0(cl) = u_0$ for any $cl \in CL$; this trace τ trivially satisfies Eq. (A.19).

(2) [**Inductive Case**: $\mathcal{X}' = \mathsf{AExectCut}(\mathcal{X}, i)$ **for** $i > 0$] Suppose that $\mathcal{X}'' = \mathsf{AExectCut}(\mathcal{X}, i-1)$ and there exists an $\mathtt{ET}$-trace τ' that satisfies Eq. (A.19). Let the set of client $CL = \{cl \mid \exists t^n_{cl}.\, t^n_{cl} \in \mathcal{X}\}$ the new transaction $t = \mathcal{X}' \setminus \mathcal{X}''$, the set of visible transactions $T = \mathsf{VIS}^{-1}_{\mathcal{X}}(t)$ and the fingerprint $\mathcal{F} = \mathcal{T}_{\mathcal{X}}(t)$. We know that $\mathcal{X}' = \mathsf{UpdateAExec}(\mathcal{X}'', T, \mathcal{F}, t)$. Note that by Def. 5.20 and Theorem 5.21, $\tau' \in \mathsf{XToTrace}(\mathcal{X}'')$. Let configuration $(\mathcal{K}'', \mathcal{U}'') = \mathsf{Last}(\tau')$. Assume the new transaction t is from client cl, that is, $t = t^{\overline{\ }}_{cl}$. Again by Def. 5.20 and Theorem 5.21, for any $\mathtt{ET}_\top$-trace $\tau \in \mathsf{XToTrace}(\mathcal{X}')$,

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

for $\mathcal{K}' = \mathsf{UpdateKV}(\mathcal{K}'', u, \mathcal{F}, t)$ and two views u, u'. The two views are defined by $u = \mathsf{GetView}(\mathcal{X}, T \cup T_{\mathrm{rd}})$ for a set of read-only transactions T_{rd}, and $u' = \mathsf{GetView}(\mathcal{X}, T')$ such that

$$T' \subseteq (\mathsf{AR}^{-1}_{\mathcal{X}})^?(t) \wedge \left(\exists t''.\, t'' = \mathsf{Min}_{\mathsf{SO}}(\{t' \mid t' \in \mathcal{X} \wedge (t, t') \in \mathsf{SO}\}) \Rightarrow T' \subseteq \mathsf{VIS}^{-1}_{\mathcal{X}}(t'')\right).$$

We need to prove that there exists an $\mathtt{ET}_\top$-trace $\tau \in \mathsf{XToTrace}(\mathcal{X}')$ that is $\mathtt{ET}$-trace. It is sufficient to find two views u, u' that satisfy $\mathcal{U}''(cl) \sqsubseteq u$ and $(\mathcal{K}'', u) \xrightarrow{\mathcal{F}}_{\mathtt{ET}} (\mathcal{K}', u')$. By the inductive hypothesis, $\mathcal{U}''(cl) \subseteq \mathsf{VIS}^{-1}_{\mathcal{X}}(t)$ and thus $\mathcal{U}''(cl) \sqsubseteq u$. Since $\mathtt{ET}$ is complete with $\mathcal{A}$, by Theorem 5.26, there exists two views u, u' that $(\mathcal{K}'', u) \xrightarrow{\mathcal{F}}_{\mathtt{ET}} (\mathcal{K}', u')$. Last, by Theorem 5.26, the invariant Eq. (A.19) is preserved for the new trace τ with respect to $\mathcal{X}'$. ■

A.6 Proofs for Section 6.1 (Verification of COPS Protocol)

Proposition A.12 (Well-defined $\mathsf{COPSInsert}$)**.** *Given a COPS store $\hat{\mathcal{K}} \in$ COPSKVS, a key $k \in$ Key and a version $\hat{\nu} \in$ COPSVer such that $\mathsf{IdOf}(\hat{\nu}) \notin \hat{\mathcal{K}}$, then the new COPS store $\mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k, \hat{\nu}\right)$ is well-formed, where the conditions are defined in Def. 6.3.*

Proof. Let $\mathcal{K}' = \mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k, \hat{\nu}\right)$ be the new store. We prove Eqs. (6.1) to (6.3) separately.

(1) [**Case**: **Eq. (6.1)**] Because $(r_0, 0)idid$ for all id, and $\mathcal{K}$ is well-formed, then $\mathcal{K}'$ satisfies Eq. (6.1).

(2) [**Case**: **Eq. (6.2)**] Because $\mathsf{IdOf}(\hat{\nu}) \notin \hat{\mathcal{K}}$, hence $\mathcal{K}'$ trivially satisfies Eq. (6.2).

(3) [**Case**: **Eq. (6.3)**] We only need to consider versions associated with k. The version list before update, $[\hat{\nu}_0, \cdots, \hat{\nu}_i, \hat{\nu}_{i+1}, \cdots, \hat{\nu}_n]$ satisfies Eq. (6.3). The new version $\hat{\nu}$ is inserted in a way preserved the order: that is, $[\hat{\nu}_0, \cdots, \hat{\nu}_i, \hat{\nu}, \hat{\nu}_{i+1}, \cdots, \hat{\nu}_n]$ where $\mathsf{IdOf}(\hat{\nu}_i) \sqsubseteq \mathsf{IdOf}(\hat{\nu}) \sqsubseteq \mathsf{IdOf}(\hat{\nu}_{i+1})$. Hence the new store $\mathcal{K}'$ satisfies Eq. (6.3).

■

Theorem A.13 (Right mover: re-fetch operations)**.** *Assume a COPS trace $\zeta \in$ COPSTrace that contains a re-fetch operation, $(cl, r, (\mathtt{R}, k, v), id, \hat{u}, \mathtt{Ref})$, that is interleaved by other operation, ι:*

$$\zeta = \zeta' \Rightarrow \left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I} \xrightarrow{(cl,r,(\mathtt{R},k,v),id,\hat{u},\mathtt{Ref})} \left(\mathcal{R}', \hat{\mathcal{U}}'\right), \mathtt{I}' \xrightarrow{\iota} \left(\mathcal{R}'', \hat{\mathcal{U}}''\right), \mathtt{I}'' \xrightarrow{\iota'} \zeta'',$$

and

$$\iota \neq (cl, r, (\mathtt{R}, k', v'), id', \hat{u}', \mathtt{Ref}) \wedge \iota \neq (cl, r, \hat{u}, \mathtt{E})$$

where all other free variables are universally quantified. Then the re-fetch operation can be moved to the right, that is,

$$\left(\zeta' \Rightarrow \left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I} \xrightarrow{\iota} \_ \xrightarrow{(cl,r,(\mathtt{R},k,v),id,\hat{u},\mathtt{Ref})} \left(\mathcal{R}'', \hat{\mathcal{U}}''\right), \mathtt{I}'' \xrightarrow{\iota'} \zeta''\right) \simeq \zeta. \tag{A.20}$$

Proof. By COPSRefetch in Fig. 6.3b, any read from the second phase does not change the store nor the context, thus

$$\zeta = \zeta' \Rightarrow \left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I} \xrightarrow{(cl,r,(\mathtt{R},k,v),id,\hat{u},\mathtt{Ref})} \left(\mathcal{R}, \hat{\mathcal{U}}'\right), \mathtt{I}' \xrightarrow{\iota} \left(\mathcal{R}'', \hat{\mathcal{U}}''\right), \mathtt{I}'' \xrightarrow{\iota'} \zeta''.$$

We perform case analysis on the label ι.

(1) [**Case**: $(cl', r', (\mathtt{W}, k', v'), id', \hat{u}')$] We immediately know $cl \neq cl'$. If $r \neq r'$, Eq. (A.20) trivially holds. Consider $r = r'$. Let $\hat{\mathcal{K}} = \mathcal{R}(r)$ and $\hat{\mathcal{K}}'' = \mathcal{R}''(r)$. It is easy to see that $\hat{\mathcal{K}} \sqsubseteq \hat{\mathcal{K}}''$, thus by Lemma A.16, we have

$$\zeta' \Rightarrow \left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I} \xrightarrow{\iota} \left(\mathcal{R}'', \hat{\mathcal{U}}''\right), \mathtt{I}^* \xrightarrow{(cl,r,(\mathtt{R},k,v),id,\hat{u},\mathtt{Ref})} \left(\mathcal{R}'', \hat{\mathcal{U}}''\right), \mathtt{I}'' \xrightarrow{\iota'} \zeta''.$$

for some program $\mathtt{I}^*$; this implies Eq. (A.20).

(2) [**Cases:** $(cl', r', \mathtt{S})$, $(cl', r', \mathtt{P})$, $(cl', r', (\mathtt{R}, k', v'), id', \hat{u}', \mathtt{Opt})$, $(cl', r', (\mathtt{R}, k', v'), id', \hat{u}', \mathtt{Opt})$, **and** $(cl', r', (\mathtt{R}, k', v'), id', \hat{u}', \mathtt{Ref})$]. First it is easy to see that $cl \neq cl'$; and these steps do not change the states of the database nor the contexts. Therefore Eq. (A.20) holds for these steps.

(3) [**Case**: $(cl', r', \hat{u}, \texttt{E})$] By COPSFinish in Fig. 6.3b, it is easy to see that $cl \neq cl'$; and this step does not change the states of the database. Because $cl \neq cl'$ the new context for cl' will not affect the cl, thus

$$\zeta' \Rightarrow \left(\mathcal{R}, \hat{\mathcal{U}}\right), \texttt{I} \xrightarrow{\iota} \left(\mathcal{R}, \hat{\mathcal{U}}''\right), \texttt{I}^* \xrightarrow{(cl', r', \hat{u}, \texttt{E})} \left(\mathcal{R}'', \hat{\mathcal{U}}''\right), \texttt{I}'' \xrightarrow{\iota'} \zeta'',$$

which implies Eq. (A.20).

(4) [**Case**: (r', id)] Let $\hat{\mathcal{K}} = \mathcal{R}(r)$ and $\hat{\mathcal{K}}'' = \mathcal{R}''(r)$. By COPSSync in Fig. 6.4b, It is easy to see that $\hat{\mathcal{K}} \sqsubseteq \hat{\mathcal{K}}''$, $\hat{\mathcal{U}}' = \hat{\mathcal{U}}''$ and $\texttt{I}' = \texttt{I}''$; thus by Lemma A.16 and the fact that COPSSync does not rely on any context nor program, we have

$$\zeta' \Rightarrow \left(\mathcal{R}, \hat{\mathcal{U}}\right), \texttt{I} \xrightarrow{\iota} \left(\mathcal{R}'', \hat{\mathcal{U}}\right), \texttt{I} \xrightarrow{(cl, r, (\texttt{R}, k, v), id, \hat{u}, \texttt{Ref})} \left(\mathcal{R}'', \hat{\mathcal{U}}''\right), \texttt{I}'' \xrightarrow{\iota'} \zeta'',$$

which implies Eq. (A.20). ■

Proposition A.14 (Monotonicity of COPS replica and client)**.** *For any COPS databases $\mathcal{R}, \mathcal{R}'$, client context environments $\hat{\mathcal{U}}, \hat{\mathcal{U}}'$, programs $\texttt{I}, \texttt{I}'$ and label ι,*

$$\begin{aligned}
&\left(\mathcal{R}, \hat{\mathcal{U}}\right), \texttt{I} \Rightarrow^* \left(\mathcal{R}', \hat{\mathcal{U}}'\right), \texttt{I}' \\
&\quad \Rightarrow \forall r \in \mathsf{Dom}(\mathcal{R}). \forall \hat{\mathcal{K}}, \hat{\mathcal{K}}' \in \mathit{COPSKVS}. \forall n, n' \in \mathbb{N}. \\
&\left(\mathcal{R}(r) = (\hat{\mathcal{K}}, n) \wedge \mathcal{R}'(r) = (\hat{\mathcal{K}}', n') \Rightarrow \hat{\mathcal{K}} \sqsubseteq \hat{\mathcal{K}}' \wedge n \leq n'\right) \\
&\qquad \wedge \forall cl \in \mathsf{Dom}\left(\hat{\mathcal{U}}\right). \forall \hat{u}, \hat{u}' \in \mathit{COPSCtx}. \\
&\qquad \left(\hat{\mathcal{U}}(cl) = (\hat{u}, _) \wedge \hat{\mathcal{U}}'(cl) = (\hat{u}', _) \Rightarrow \hat{u} \subseteq \hat{u}'\right).
\end{aligned} \tag{A.21}$$

Proof. We prove by induction on the length of the trace n.

[**Base Case**: $n = 0$] Eq. (A.21) is trivially true as $\mathcal{R} = \mathcal{R}'$ and $\hat{\mathcal{U}} = \hat{\mathcal{U}}'$.

[**Base Case**: $n > 0$] Assume $\left(\mathcal{R}, \hat{\mathcal{U}}\right), \texttt{I} \Rightarrow^{(n-1)} \left(\mathcal{R}'', \hat{\mathcal{U}}''\right), \texttt{I}'' \xrightarrow{\iota} \left(\mathcal{R}', \hat{\mathcal{U}}'\right), \texttt{I}'$. By inductive hypothesis, we know $\mathcal{R}(r)_{|0} \sqsubseteq \mathcal{R}'(r)_{|0}$ and $\mathcal{R}(r)_{|1} \leq \mathcal{R}'(r)_{|1}$ for $r \in \mathsf{Dom}(\mathcal{R})$. Consider the label ι. If $\iota = (cl, r, (\texttt{W}, k, v), id, \hat{u})$, by COPSWrite, we have $\mathcal{R}' = \mathcal{R}''\left[r \mapsto \left(\hat{\mathcal{K}}^*, \mathcal{R}''(r)_{|1} + 1\right)\right]$, for $\hat{\mathcal{K}}^* = \mathsf{COPSInsert}\left(\mathcal{R}''(r)_{|0}, k, \hat{\nu}\right)$ for a version $\hat{\nu}$ and, thus Eq. (A.21). If $\iota = (r, id)$, by COPSSync we have $\mathcal{R}' = \mathcal{R}''\left[r \mapsto \left(\hat{\mathcal{K}}^*, m\right)\right]$ such that $m = \mathsf{Max}\left(\mathcal{R}''(r)_{|1}, m^*\right)$ where $id = id[cl^*][r^*](m^*)$, and $\hat{\mathcal{K}}' = \mathsf{COPSInsert}\left(\mathcal{R}''(r)_{|0}, k, \hat{\nu}\right)$ for a version $\hat{\nu}$, and, thus Eq. (A.21). For the rest cases, they do not change any COPS store and local time; by inductive hypothesis, Eq. (A.21) holds. ■

Theorem A.15 (Left mover: out-of-order write)**.** *Assume a COPS trace $\zeta \in \mathit{COPSTrace}$ such that*

$$\zeta = \zeta' \Rightarrow \left(\mathcal{R}, \hat{\mathcal{U}}\right), \texttt{I} \xrightarrow{\iota} \left(\mathcal{R}', \hat{\mathcal{U}}'\right), \texttt{I}' \xrightarrow{(cl, r, (\texttt{W}, k, v), id, \hat{u})} \left(\mathcal{R}'', \hat{\mathcal{U}}''\right), \texttt{I}'' \xrightarrow{\iota'} \zeta''$$

for a write operation $(\mathtt{W}, k, v)$ *from a client* cl *with context* $\hat{u}$ *and a replica* r*. If the step,* ι*, from a different client* cl' *before the write operation satisfies:*

$$\begin{aligned}&(\iota = (cl', r', (\mathtt{W}, k', v'), id', \hat{u}') \wedge (r \neq r'))\\ &\vee (\iota = (cl', r', (\mathtt{R}, k', v'), id', \hat{u}', \mathtt{Opt}) \wedge (r = r' \wedge k' = k \Rightarrow id \sqsubseteq id'))\\ &\vee \iota = (cl', r', (\mathtt{R}, k', v'), id', \hat{u}', \mathtt{Ref}) \vee \iota = (cl', r', \mathtt{S})\\ &\vee \iota = (cl', r', \mathtt{P}) \vee \iota = (cl', r', \hat{u}', \mathtt{E}) \vee \iota = (r', id'),\end{aligned} \tag{A.22}$$

then the write operation can be moved left, that is,

$$\left(\zeta' \Rightarrow \left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I} \xrightarrow{(cl, r, (\mathtt{W}, k, v), id, \hat{u})} \_ \xrightarrow{\iota} \left(\mathcal{R}'', \hat{\mathcal{U}}''\right), \mathtt{I}'' \xrightarrow{\iota'} \zeta''\right) \simeq \zeta. \tag{A.23}$$

Proof. Note that ι are all annotated with replica r'. If $r \neq r'$ by COPSClient in Fig. 6.4b Eq. (A.23) trivially holds. Now consider $r = r'$. We prove Eq. (A.23) by case analysis on label ι.

(1) [**Case**: $(cl', r, (\mathtt{R}, k', v'), id', \hat{u}', \mathtt{Opt})$] For this case, cl and cl' must be two distinct clients. If $k \neq k'$, Eq. (A.23) trivially holds. Consider $k = k'$ and $id \sqsubseteq id'$ Let $\hat{\nu} = (v, id, \hat{u})$ and $\hat{\nu}' = (v', id', \hat{u}')$. Let $\hat{\mathcal{K}} = \mathcal{R}(r)$ and $\hat{\mathcal{K}}'' = \mathcal{R}''(r)$. By COPSOpt in Fig. 6.3b, $\hat{\mathcal{K}} = \mathcal{R}(r) = \mathcal{R}'(r)$, and $\hat{\nu}' = \hat{\mathcal{K}}\left(k, \left|\hat{\mathcal{K}}(k)\right| - 1\right)$. By COPSWrite in Fig. 6.2b, we know $\hat{\mathcal{K}}'' = \mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k, \hat{\nu}'\right)$; since $id \sqsubseteq id'$, the version $\hat{\nu}$ is inserted in the middle of $\hat{\mathcal{K}}(k)$. Therefore we know that the last version of $\hat{\mathcal{K}}''(k)$ is still $\hat{\nu}'$, that is, $\hat{\nu}' = \hat{\mathcal{K}}''(k, \left|\hat{\mathcal{K}}''(k)\right| - 1)$. This means that the write operation does not interfere with the previous optimistic read, thus Eq. (A.23) holds.

(2) [**Case**: $(cl', r, (\mathtt{R}, k', v'), id', \hat{u}', \mathtt{Opt})$] For this case, cl and cl' must be two distinct clients. Let $\hat{\nu} = (v, id, \hat{u})$ and $\hat{\nu}' = (v', id', \hat{u}')$. Let $\hat{\mathcal{K}} = \mathcal{R}(r)$ and $\hat{\mathcal{K}}'' = \mathcal{R}''(r)$. By COPSRefetch in Fig. 6.3b, $\hat{\mathcal{K}} = \mathcal{R}(r) = \mathcal{R}'(r)$; and by COPSWrite in Fig. 6.2b, $\hat{\mathcal{K}}'' = \mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k', \hat{\nu}'\right)$. It is easy to see that $\hat{\mathcal{K}} \sqsubseteq \hat{\mathcal{K}}''$ defined in Lemma A.16. By Lemma A.16 we know that Eq. (A.23) holds.

(3) [**Cases**: $(cl', r, \mathtt{S})$, $(cl', r, \mathtt{P})$ **and** $(cl', r, \hat{u}', \mathtt{E})$] Given the rules presented in Fig. 6.3b, these steps are local to the client cl' without any interaction to the key-value store; thus Eq. (A.23) holds.

(4) [**Case**: (r, id')] A new version $\hat{\nu}'$ indexed by id' arrives to the replica r. Let $\hat{\nu} = (v, id, \hat{u})$. Let $\hat{\mathcal{K}} = \mathcal{R}(r)$, $\hat{\mathcal{K}}' = \mathcal{R}'(r)$ and $\hat{\mathcal{K}}'' = \mathcal{R}''(r)$. By COPSSync in Fig. 6.4b, we know $\hat{\mathcal{K}}' = \mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k', \hat{\nu}'\right)$, $\hat{\mathcal{K}}'' = \mathsf{COPSInsert}\left(\hat{\mathcal{K}}', k, \hat{\nu}\right)$ and $id \neq id'$. By the definition of $\mathsf{COPSInsert}$, there exists $\hat{\mathcal{K}}^*$ such that

$$\hat{\mathcal{K}}^* = \mathsf{COPSInsert}\left(\hat{\mathcal{K}}, k, \hat{\nu}\right) \wedge \hat{\mathcal{K}}'' = \mathsf{COPSInsert}\left(\hat{\mathcal{K}}^*, k, \hat{\nu}'\right),$$

and therefore, a new cops database $\mathcal{R}^*$ such that

$$\mathcal{R}^* = \mathcal{R}\left[r \mapsto \hat{\mathcal{K}}^*\right] \wedge \mathcal{R}'' = \mathcal{R}^*\left[r \mapsto \hat{\mathcal{K}}''\right].$$

The rule COPSSync does not depend nor change the context environment, thus we have the proof for Eq. (A.23). ■

Lemma A.16 (Re-fetching version on a larger COPS store)**.** *Given two COPS stores $\hat{\mathcal{K}}, \hat{\mathcal{K}}'$ such $\hat{\mathcal{K}} \sqsubseteq \hat{\mathcal{K}}'$, then*

$$\left(\hat{\mathcal{K}}, \hat{u}, n\right), \texttt{read}\,(K) : (\hat{\mathcal{V}}, D, \hat{\mathcal{V}}') \xrightarrow{(cl,r,(\texttt{R},k_i,v_i),id_i,d_i,\texttt{Ref})} \left(\hat{\mathcal{K}}, \hat{u}, n\right), \texttt{read}\,(K) : (\hat{\mathcal{V}}, D, \hat{\mathcal{V}}'')$$
$$\Rightarrow \left(\hat{\mathcal{K}}', \hat{u}, n\right), \texttt{read}\,(K) : (\hat{\mathcal{V}}, D, \hat{\mathcal{V}}') \xrightarrow{(cl,r,(\texttt{R},k_i,v_i),id_i,d_i,\texttt{Ref})} \left(\hat{\mathcal{K}}', \hat{u}, n\right), \texttt{read}\,(K) : (\hat{\mathcal{V}}, D, \hat{\mathcal{V}}'')$$

Proof. Depending on $\hat{\mathcal{V}}_{|i}$, we have two possible cases.

(1) [**Case**: $\mathsf{IdOf}\left(\hat{\mathcal{V}}_{|i}\right) \sqsubseteq id_i$] For this case, there exists an version $\hat{\mathcal{K}}\,(k_i, m) = (v_i, id_i, d_i)$ for some index m. Given a new store $\hat{\mathcal{K}}'$ such that $\hat{\mathcal{K}} \sqsubseteq \hat{\mathcal{K}}'$, this version must be also included in $\hat{\mathcal{K}}'$, thus $\hat{\mathcal{K}}'(k_i, m') = (v_i, id_i, d_i)$ for some m'; therefore we have the proof.

(2) [**Case**: $\mathsf{IdOf}\left(\hat{\mathcal{V}}_{|i}\right) = id_i$] For this case, the state of $\hat{\mathcal{K}}$ is irrelevant. ■

Proposition A.17 (Fresh multiple-read transaction identifiers)**.** *Given an annotated normalised COPS trace $\zeta \in$ ANCOPSTrace, the annotated multiple-read transaction identifiers must be fresh with respect the key being read, that is,*

$$\zeta = \zeta' \Rightarrow \left(\mathcal{R}, \hat{\mathcal{U}}\right), \texttt{I} \xrightarrow{\left(cl,r,(\texttt{R},k,v),\hat{t},d,\texttt{Ref}\right),\hat{t}'} \zeta''$$
$$\wedge\, \forall r^* \in \mathit{COPSRep}.\, \forall cl^* \in \mathit{CID}.\, \forall v^* \in \mathit{Value}.\, \forall d^* \in \mathit{COPSDep}.$$
$$\zeta'' = \cdots \xrightarrow{\left(cl^*,r^*,(\texttt{R},k,v^*),\hat{t}^*,d^*,\texttt{Ref}\right),\hat{t}''} \cdots \Rightarrow \hat{t}' \neq \hat{t}''$$

Proof. Suppose a replica r^*, a client cl^*, a value v^* and dependency set d^* such that

$$\zeta'' = \cdots \xrightarrow{\left(cl^*,r^*,(\texttt{R},k,v^*),\hat{t}^*,d^*,\texttt{Ref}\right),\hat{t}''} \cdots.$$

If $r \neq r^*$ or $cl \neq cl^*$, by the definition of ANCOPSTrace (especially COPSToExt), it is trivial that $\hat{t} \neq \hat{t}''$. Consider $r = r^*$ or $cl = cl^*$. Because a client must provide a unique set of key K when calling `read` by the rules in Fig. 6.3b, it means that

$$\zeta'' = \cdots \xrightarrow{(cl,r,d',\texttt{Ref}),\hat{t}'} \cdots \xrightarrow{\left(cl^*,r^*,(\texttt{R},k,v^*),\hat{t}^*,d^*,\texttt{Ref}\right),\hat{t}''} \cdots.$$

For this case, by the definition of ANCOPSTrace (especially COPSToExt), it must be the case that $\hat{t} \neq \hat{t}''$. ■

Proposition A.18 (Appending write operations)**.** *Given an annotated normalised COPS trace ζ, for every write operation, $\iota = \left(cl, r, (\mathtt{W}, k, v), \hat{t}, d\right)$, the version identifier, $\hat{t}$, is strictly greater than all version identifiers included in all replicas: that is,*

$$\begin{aligned}\zeta = \zeta' \Rightarrow \left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I} \xrightarrow{\left(cl, r, (\mathtt{W}, k, v), \hat{t}, d\right)} \left(\mathcal{R}', \hat{\mathcal{U}}'\right), \mathtt{I}' \Rightarrow \cdots \\ \wedge \forall r' \in \mathit{COPSRep}.\, \forall \hat{\mathcal{K}}' \in \mathit{COPSKVS}.\, \forall k' \in \mathit{Key}.\, \forall i' \in \mathbb{N}. \\ \mathcal{R}(r') = (\hat{\mathcal{K}}', \_) \wedge 0 \leq i < \hat{\mathcal{K}}(k') \Rightarrow \mathsf{WriterOf}\left(\hat{\mathcal{K}}'(k', i')\right) \sqsubset \hat{t}\end{aligned}$$

Proof. Suppose a replica r' with key-value store $\hat{\mathcal{K}}'$ ($\mathcal{R}(r') = (\hat{\mathcal{K}}', \_)$), a key k' and an index i' such that $0 \leq i < \hat{\mathcal{K}}(k')$. Let the $\hat{t}' = \mathsf{WriterOf}\left(\hat{\mathcal{K}}(k', i')\right)$. There must exist a step with label ι' with some client cl' in the trace ζ' that committed this version:

$$\zeta' = \cdots \xrightarrow{\iota'} \cdots \wedge \iota' = \left(cl, r, (\mathtt{W}, k', v'), \hat{t}', d\right)$$

for some value v' and dependent set d. Since an annotated normalised trace ζ is also in its normalised form, that is, $\mathtt{NCOPSTrace}(\zeta)$, and the new kv-store $\mathcal{R}'(r')_{|0}$ must be a well-formed COPS key-value store, therefore $\hat{t}' \sqsubset \hat{t}$. ■

Proposition A.19 (Well-defined COPSToKVS)**.** *Given an annotated normalised COPS trace ζ, the kv-store $\mathcal{K} = \mathsf{COPSToKVS}(\zeta)$ is well-formed, that is, $\mathtt{WfKvs}(\mathcal{K})$. The kv-store contains and only contains versions in the last configuration $((\mathcal{R}, \hat{\mathcal{U}}), \mathtt{I}) = \mathsf{Last}(\zeta)$: that is,*

$$\begin{aligned}&\forall k \in \mathit{Key}.\, \forall i \in \mathbb{N}.\, \forall v \in \mathit{Value}.\, \forall \hat{t} \in \mathit{COPSTxID}.\, \mathcal{K}(k, i) = (v, \hat{t}, \_) \\ &\quad \Rightarrow \exists r \in \mathit{COPSRep}.\, \exists i' \in \mathbb{N}.\, \exists r, n.\, \hat{t} = \hat{t}_{\_}^{(n,r,\_)} \mathcal{R}(r)(k, i') = (v, (n, r), \_), \qquad \text{(A.24)} \\ &\forall k \in \mathit{Key}.\, \forall i \in \mathbb{N}.\, \forall v \in \mathit{Value}.\, \forall r, n. \\ &\quad \mathcal{R}(r)(k, i) = (v, (n, r), \_) \Rightarrow \exists i' \in \mathbb{N}.\, \exists cl.\, \mathcal{K}(k, i') = (v, \hat{t}_{cl}^{(n,r,0)}, \_). \qquad \text{(A.25)}\end{aligned}$$

Proof. We prove this by induction on the length of trace ζ.

(1) [**Cases**: $\zeta = \Theta_0$] By the definition of COPSToKVS, the kv-store $\mathcal{K}_0 = \mathsf{COPSToKVS}(\Theta_0)$ is trivially well-formed. By the definition of Θ_0, Eqs. (A.24) and (A.25) hold.

(2) [**Cases**: $\zeta = \zeta' \xrightarrow{\iota} \left(\mathcal{R}, \hat{\mathcal{U}}\right), \mathtt{I}$] Let $\mathcal{K}' = \mathsf{COPSToKVS}(\zeta')$; by inductive hypothesis, the kv-store $\mathcal{K}'$ is a well-formed kv-store and satisfies Eqs. (A.24) and (A.25). Consider label ι.

(i) [**Cases**: $\iota = (cl, r, (\mathtt{W}, k, v), (n, r), \hat{u}), \hat{t}_{cl}^{(n,r,0)}$] By the definition of COPSToKVS, the new kv-store $\mathcal{K} = \mathsf{COPSToKVS}(\zeta)$ is given by $\mathcal{K} = \mathcal{K}'\left[k \mapsto \mathcal{K}'(k) :: \left[(v, \hat{t}_{cl}^{(n,r,0)}, \emptyset)\right]\right]$. By COPSWrite, the transaction identifier for the new version must be fresh, that is, $\hat{t} \notin \mathcal{K}$, which implies $\mathtt{WfKvs}(\mathcal{K})$. Let $(\mathcal{R}, \hat{\mathcal{U}}) = \mathsf{Last}(\zeta)$ and $(\hat{\mathcal{K}}, \_) = \mathcal{R}(r)$;

The new version must be included in $\hat{\mathcal{K}}$: $\hat{\mathcal{K}}\left(k, \left|\hat{\mathcal{K}}\left(k\right)\right| - 1\right) = (v, (n, r), d)$ for some dependent set d, which means Eqs. (A.24) and (A.25).

(ii) [**Cases**: $\iota = \left(\left(cl, r, (\mathtt{R}, k, v), \hat{t}, \hat{u}, \_\right), \hat{t}_{cl}^{(n,r,m)}\right)$] By the definition of $\mathsf{COPSToKVS}$, the new kv-store $\mathcal{K} = \mathsf{COPSToKVS}\left(\zeta\right)$ is given by:

$$\mathcal{K} = \mathcal{K}'\left[k \mapsto \mathcal{K}'(k)\left[i \mapsto \left(v, \hat{t}, T \cup \hat{t}_{cl}^{(n,r,m)}\right)\right]\right]$$

and $\mathcal{K}'(k)_{|i} = (v, \hat{t}, T)$ for some i, v, T. By Prop. A.17, the transaction identifier $\hat{t}_{cl}^{(n,r,m)}$ must be fresh with respect to the key k, that is $\hat{t}_{cl}^{(n,r,m)} \notin \mathcal{K}'(k, i')$ for all index i' in range. This implies $\mathtt{WfKvs}\left(\mathcal{K}\right)$. There is no new version. Therefore the new kv-store $\mathcal{K}$ satisfies Eqs. (A.24) and (A.25).

(iii) [**Cases**: **other** ι] By the definition of $\mathsf{COPSToKVS}$, the new kv-store is given by:

$$\mathcal{K} = \mathsf{COPSToKVS}\left(\zeta\right)$$

and by inductive hypothesis it is a well-formed kv-store and satisfies Eqs. (A.24) and (A.25). ■

Proof sketch. It is easy to prove $\mathtt{WfKvs}\left(\mathcal{K}\right)$ and Eqs. (A.24) and (A.25) by induction on the trace ζ. Note that both single-write and multiple-read transaction identifiers are unique, (Prop. A.17), which is the key to prove $\mathtt{WfKvs}\left(\mathcal{K}\right)$. Eqs. (A.24) and (A.25) can be proven by the definition of $\mathsf{COPSToKVS}$, since the function only replays all transactions in the COPS trace. The full detail is given in appendix A.6 on page 209. ■

Proposition A.20 (Well-defined $\mathsf{COPSViews}$)**.** *Assume an annotated normalised COPS trace ζ and the last configuration $((\mathcal{R}, \hat{\mathcal{U}}), \_) = \mathsf{Last}\left(\zeta\right)$. Let $\mathcal{K} = \mathsf{COPSToKVS}\left(\zeta\right)$ and $\mathcal{U} = \mathsf{COPSViews}\left(\mathcal{R}, \hat{\mathcal{U}}\right)$. Then every view in $\mathcal{U}$ is a well-formed view on $\mathcal{K}$: that is, $\mathcal{U}\left(cl\right) \in \textsc{ViewOn}\left(\mathcal{K}\right)$ for all clients $cl \in \mathsf{Dom}\left(\mathcal{U}\right)$.*

Proof. Assume a client cl and the view $u = \mathcal{U}\left(cl\right)$ for the client cl. By the definition of $\mathsf{COPSViews}$, Eqs. (4.5) and (4.6) are trivially true. Consider keys k, k' and indices i, i' such that $i \in u\left(k\right)$ and $\mathsf{WriterOf}\left(\mathcal{K}\left(k, i\right)\right) = \mathsf{WriterOf}\left(\mathcal{K}\left(k', i'\right)\right)$. By Eq. (6.2) $k = k'$ and $i = i'$ therefore Eq. (4.7) holds. ■

Theorem 6.19 (COPS causal consistency)**.** *Given an annotated normalised COPS trace $\zeta \in$ ANCOPSTrace, the kv-store traces, $\eta = \mathsf{COPSToKVTrace}\left(\zeta\right)$, can be obtained under $\mathtt{ET}_{\mathtt{CC}}$.*

Proof. Recall the definition of $\texttt{CC}$ that for kv-stores $\mathcal{K}, \mathcal{K}'$, views u, u' and fingerprint $\mathcal{F}$,

$$u \sqsubseteq u' \tag{A.26}$$

$$\forall t \in \mathcal{K}' \setminus \mathcal{K}.\, \forall k \in \textsc{Key}.\, \forall i \in \mathbb{N}.\, \mathsf{WriterOf}\left(\mathcal{K}'(k,i)\right) \xrightarrow{\mathsf{SO}^?} t \Rightarrow i \in u'(k) \tag{A.27}$$

$$\texttt{PreClosed}\left(\mathcal{K}, u, \mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO}\right) \tag{A.28}$$

We first show that every step in the trace η satisfies Eq. (A.26); then we show that every step *preserves* Eqs. (A.27) and (A.28) for all views included in the view environment. We now prove by induction on the length of the trace η.

(1) [**Base Case**: $\eta = \left(\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}\right), \texttt{P}$] By the definition of $\mathcal{K}_0$ and $\mathcal{U}_0$, It is trivial that Eqs. (A.27) and (A.28) hold.

(2) [**Inductive Case**: $\eta = \eta' \xrightarrow{\iota}_{\top} \left(\mathcal{K}, \mathcal{U}, \mathcal{E}\right), \texttt{P}$] Assume that $\eta = \mathsf{COPSToKVTrace}\left(\zeta\right)$ and $\texttt{CC}$-trace $\eta' = \mathsf{COPSToKVTrace}\left(\zeta'\right)$. Consider the label ι.

(i) [**Case**: $\iota = \left(cl, u, \{(\texttt{W}, k, v)\}\right)$] By the definition of $\mathsf{COPSToKVTrace}$, traces ζ, ζ' satisfy the following property

$$\begin{aligned} u = \mathsf{COPSViews}\left(\zeta'\right)\left(cl\right) \wedge \zeta = \zeta' \xrightarrow{(cl,r,(\texttt{W},k,v),(n,r),d),\hat{t}_{cl}^{(n,r,0)}} \left(\mathcal{R}, \hat{\mathcal{U}}\right), \texttt{I} \\ \wedge\, \mathsf{Last}\left(\zeta'\right) = \left(\mathcal{R}', \hat{\mathcal{U}}'\right) \\ \wedge\, \hat{\mathcal{U}}' = \hat{\mathcal{U}}\left[cl \mapsto \left(\hat{\mathcal{U}}\left(cl\right)_{|0} \cup \left\{\left(k, \hat{t}_{cl}^{(n,r,0)}\right)\right\}, r\right)\right] \end{aligned} \tag{A.29}$$

Let $u' = \mathsf{COPSViews}\left(\zeta\right)\left(cl\right) = \mathcal{U}\left(cl\right)$ be the view after update. By Eq. (A.29) and the definition of $\mathsf{COPSViews}$, the view must increase as $u \sqsubseteq u'$ and thus Eq. (A.26); also the new view u' must contains the new version written by $\hat{t}_{cl}^{(n,r,0)}$ and thus Eq. (A.27). By the definition of $\mathsf{COPSViews}$ and Prop. 6.20, it follows Eq. (A.28).

(ii) [**Case**: $\iota = \left(cl, u, \{(\texttt{R}, k_0, v_0), \cdots, (\texttt{R}, k_n, v_n)\}\right)$] By the definition of $\mathsf{COPSToKVTrace}$, traces ζ, ζ' satisfy the following property

$$\begin{aligned} u = \mathcal{U}\left(cl\right) \\ \wedge\, \zeta = \zeta' \xrightarrow{(cl,r,(\texttt{R},k_0,v_0),id_0,(n_0,r_0),\texttt{Ref}),\hat{t}} \cdots \xrightarrow{(cl,r,(\texttt{R},k_m,v_m),(n_m,r_m),\hat{u}_n,\texttt{Ref}),\hat{t}} \\ \zeta = \cdots \xrightarrow{(cl,r,\hat{u}',\texttt{E}),\hat{t}} \left(\mathcal{R}, \hat{\mathcal{U}}\right), \texttt{I} \\ \wedge\, \mathsf{Last}\left(\zeta'\right) = \left(\mathcal{R}', \hat{\mathcal{U}}'\right) \wedge \hat{\mathcal{U}}' = \hat{\mathcal{U}}\left[cl \mapsto \left(\hat{\mathcal{U}}\left(cl\right)_{|0} \cup \hat{u}', r\right)\right] \end{aligned} \tag{A.30}$$

By Eq. (A.30) and the definition of $\mathsf{COPSViews}$, the view before and after read are the same view u and thus Eq. (A.26); since there is no new version and view cannot lost any version, thus Eq. (A.27) holds. By rule COPSFinish, it follows that $(k_i, (n_i, r_i)) \in \hat{u}'$ for index i such that $0 \leq i \leq m$. Then by Prop. 6.20, for any

$k, j,$

$$\mathsf{WriterOf}\left(\mathcal{K}\left(k,j\right)\right) \xrightarrow{\left(\mathsf{WR}_{\mathcal{K}}\cup\mathsf{SO}\right)^{+}} \hat{t}_{cl_i}^{(n_i,r_i,0)} \Rightarrow j \in u\left(k\right), \tag{A.31}$$

Given above, Eq. (A.31), the closure property Eq. (A.28) holds. ■

A.7 Proofs for Section 6.2 (Verification of Clock-SI Protocol)

Proposition 6.35 (Right mover: Clock-SI internal read and write steps)**.** *Assume a Clock-SI trace $\phi \in A\textsc{ClockTrace}$, and two adjacent transitions with labels ι, ι' such that*

$$\begin{aligned}\phi = \phi' \xrightarrow{\iota''} \left(\mathcal{R},\tilde{\mathcal{U}},\mathcal{E}\right), \mathtt{I} \xrightarrow{\iota} \left(\mathcal{R}',\tilde{\mathcal{U}}',\mathcal{E}'\right), \mathtt{I}' \xrightarrow{\iota'} \phi'' \\ \wedge \left(\iota = \left(cl, r, \left(\mathtt{W}, k, v\right), n, n''\right) \vee \iota = \left(cl, r, \left(\mathtt{R}, k, v\right), n, n''\right)\right) \\ \wedge \forall l, k', v', n', n'''.\, \iota' \neq \left(cl, r, \left(l, k', v'\right), n', n'''\right) \wedge \iota' \neq \left(cl, r, n''', \mathtt{E}\right)\end{aligned}$$

where the rest free variables are universally quantified. The transition labelled by ι can be moved to the right, that is, there exists a new equivalent trace ϕ_1 such that

$$\phi_1 = \phi' \xrightarrow{\iota''} \_ \xrightarrow{\iota'} \_ \xrightarrow{\iota} \phi'' \wedge \phi \simeq \phi_1.$$

Proof. Assume a trace ϕ such that $\phi = \phi' \xrightarrow{\iota''} \left(\mathcal{R},\tilde{\mathcal{U}},\mathcal{E}\right), \mathtt{I} \xrightarrow{\iota} \left(\mathcal{R}',\tilde{\mathcal{U}}',\mathcal{E}'\right), \mathtt{I}' \xrightarrow{\iota'} \phi''$ and assume ι such that $\iota = \left(cl, r, \left(\mathtt{W}, k, v\right), n\right) \vee \iota = \left(cl, r, \left(\mathtt{R}, k, v\right), n\right)$ where all other free variables are universal quantified. By the CSIWrite, CSIReadL and CSIReadS rules, thus there exists a stack s and a Clock-SI runtime command $\mathtt{R}$,

$$\mathcal{R} = \mathcal{R}' \wedge \tilde{\mathcal{U}} = \tilde{\mathcal{U}}' \wedge \mathcal{E}' = \mathcal{E}\left[cl \mapsto s\right] \wedge \mathtt{P}' = \mathtt{P}\left[cl \mapsto \mathtt{R}\right].$$

This means that $\phi = \phi' \xrightarrow{\iota''} \left(\mathcal{R},\tilde{\mathcal{U}},\mathcal{E}\right), \mathtt{I} \xrightarrow{\iota} \left(\mathcal{R},\tilde{\mathcal{U}},\mathcal{E}\left[cl \mapsto s\right]\right), \mathtt{I}\left[cl \mapsto \mathtt{R}\right] \xrightarrow{\iota'} \phi''$. We perform case analysis on ι'.

(1) [**Case**: $\iota' = \left(r', cl', n', \mathtt{S}\right)$] We immediately have $cl \neq cl'$. By CSISnapshot rule, there exists a Clock-SI runtime command $\mathtt{R}'$ such that

$$\left(\mathcal{R},\tilde{\mathcal{U}},\mathcal{E}\left[cl \mapsto s\right]\right), \mathtt{I}\left[cl \mapsto \mathtt{R}\right] \xrightarrow{\left(r', cl', n', \mathtt{S}\right)} \left(\mathcal{R},\tilde{\mathcal{U}},\mathcal{E}\left[cl \mapsto s\right]\right), \mathtt{I}\left[cl \mapsto \mathtt{R}\right]\left[cl' \mapsto \mathtt{R}'\right].$$

Note that $\mathtt{I}\left[cl \mapsto \mathtt{R}\right]\left[cl' \mapsto \mathtt{R}'\right] = \mathtt{I}\left[cl' \mapsto \mathtt{R}'\right]\left[cl \mapsto \mathtt{R}\right]$. This means that

$$\phi \simeq \left(\phi' \xrightarrow{\iota''} \left(\mathcal{R},\tilde{\mathcal{U}},\mathcal{E}\right), \mathtt{I} \xrightarrow{\iota'} \left(\mathcal{R},\tilde{\mathcal{U}},\mathcal{E}\right), \mathtt{I}\left[cl' \mapsto \mathtt{R}'\right] \xrightarrow{\iota} \phi''\right).$$

(2) [**Case**: $\iota' = (r', cl', (l', k', v'), n')$] We immediately have $cl \neq cl'$ and $l \in \{\mathtt{W}, \mathtt{R}, \mathtt{P}\}$. By CSIWrite (for W), CSIReadL, CSIReadS (for R) and CSIPrepare (for P) rules, there exists a stack s' and a Clock-SI runtime command $\mathtt{R}'$ such that

$$\left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}\left[cl \mapsto s\right]\right), \mathtt{I}\left[cl \mapsto \mathtt{R}\right] \xrightarrow{(r', cl', (l', k', v'), n')} \left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}\left[cl \mapsto s\right]\left[cl' \mapsto s'\right]\right), \mathtt{I}\left[cl \mapsto \mathtt{R}\right]\left[cl' \mapsto \mathtt{R}'\right].$$

Note that

$$\mathcal{E}\left[cl \mapsto s\right]\left[cl' \mapsto s'\right] = \mathcal{E}\left[cl' \mapsto s'\right]\left[cl \mapsto s\right] \wedge \mathtt{I}\left[cl \mapsto \mathtt{R}\right]\left[cl' \mapsto \mathtt{R}'\right] = \mathtt{I}\left[cl' \mapsto \mathtt{R}'\right]\left[cl \mapsto \mathtt{R}\right].$$

This means that

$$\phi \simeq \left(\phi' \xrightarrow{\iota''} \left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}\right), \mathtt{I} \xrightarrow{\iota'} \left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}\left[cl' \mapsto s'\right]\right), \mathtt{I}\left[cl' \mapsto \mathtt{R}'\right] \xrightarrow{\iota} \phi''\right).$$

(3) [**Case**: $\iota' = (r', cl', n', \mathtt{E})$] We immediately have $cl \neq cl'$. By CSICommit rule there exists a new Clock-SI database $\mathcal{R}'$, and a Clock-SI runtime command $\mathtt{R}'$ such that

$$\left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}\left[cl \mapsto s\right]\right), \mathtt{I}\left[cl \mapsto \mathtt{R}\right] \xrightarrow{(r', cl', (l', k', v'), n')} \left(\mathcal{R}', \tilde{\mathcal{U}}\left[cl' \mapsto n'\right], \mathcal{E}\left[cl \mapsto s\right]\right), \mathtt{I}\left[cl \mapsto \mathtt{R}\right]\left[cl' \mapsto \mathtt{R}'\right] \wedge \mathcal{R} \sqsubseteq \mathcal{R}'.$$

Note that $\mathtt{I}\left[cl \mapsto \mathtt{R}\right]\left[cl' \mapsto \mathtt{R}'\right] = \mathtt{I}\left[cl' \mapsto \mathtt{R}'\right]\left[cl \mapsto \mathtt{R}\right]$. Then by $\mathcal{R} \sqsubseteq \mathcal{R}'$ and Prop. A.21, the transition labelled by ι can be proceeded on $\mathcal{R}$. This means that

$$\phi \simeq \left(\phi' \xrightarrow{\iota''} \left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}\right), \mathtt{I} \xrightarrow{\iota'} \left(\mathcal{R}', \tilde{\mathcal{U}}\left[cl' \mapsto n'\right], \mathcal{E}\right), \mathtt{I}\left[cl' \mapsto \mathtt{R}'\right] \xrightarrow{\iota} \phi''\right). \qquad \blacksquare$$

Proposition A.21 (Read and write steps on a larger Clock-SI database)**.** *Assume a read or write transition*

$$\left(\mathcal{R}, n, s\right), \mathtt{R} \xrightarrow{\iota} \left(\mathcal{R}, n', s'\right), \mathtt{R}' \wedge \left(\iota = \left(cl, r, \left(\mathtt{W}, k, v\right), n''\right) \vee \iota = \left(cl, r, \left(\mathtt{R}, k, v\right), n''\right)\right).$$

Given a Clock-SI database $\mathcal{R}'$ such that $\mathcal{R} \sqsubseteq \mathcal{R}'$, the same transition can be proceeded on $\mathcal{R}'$, that is,

$$\left(\mathcal{R}, n, s\right), \mathtt{R} \xrightarrow{\iota} \left(\mathcal{R}, n', s'\right), \mathtt{R}'$$

Proof. We consider rules CSIWrite, CSIReadL and CSIReadS separately.

(1) [**Cases**: **CSIWrite and CSIReadL**] Given the premise of CSIWrite and CSIReadL, these two transitions do not depend on the state of $\mathcal{R}$.

(2) [**Case**: **CSIReadS**] We have $\iota = \left(cl, r, \left(\mathtt{R}, k, v\right), n\right)$ for some r, cl, k, v, n. Let r' be the shard defined by $r' = \mathsf{ShardOf}_{\mathcal{R}}\left(k\right)$ and $\left(\tilde{\mathcal{K}}, n_1\right) = \mathcal{R}\left(r'\right)$. By the premise of CSIReadS,

the shard local time must be greater than the snapshot time, that is $n_1 > n''$. Let $\tilde{\mathcal{V}} = \left\{\tilde{\nu} \;\middle|\; \exists i.\, \tilde{\mathcal{K}}\,(k,i) = \tilde{\nu} \wedge \mathsf{TimeOf}\,(\tilde{\nu}) \leq n''\right\}$ be the set of versions of k that have time-stamp smaller than the snapshot time n''. By the premise of CSIReadS, $(kv, \_) = \mathsf{Max}\left(\tilde{\mathcal{V}}\right)$. Because the hypothesis that $\mathcal{R} \sqsubseteq \mathcal{R}'$. Let $Tuple\tilde{\mathcal{K}}', n_2 = \mathcal{R}'(r')$. We have $\tilde{\mathcal{K}} \sqsubseteq \tilde{\mathcal{K}}'$ and $n_1 \leq n_2$. Therefore $n_2 > n''$ and $\tilde{\mathcal{V}} = \left\{\tilde{\nu} \;\middle|\; \exists i.\, \tilde{\mathcal{K}}'(k,i) = \tilde{\nu} \wedge \mathsf{TimeOf}\,(\tilde{\nu}) \leq n\right\}$, which means the same transition can be proceeded on $\mathcal{R}'$. In other words,

$$(\mathcal{R}, n, s)\,, \mathtt{R} \xrightarrow{\iota} (\mathcal{R}, n', s')\,, \mathtt{R}' \qquad \blacksquare$$

Proposition 6.36 (Left mover: Clock-SI preparation and commit steps)**.** *Assume a Clock-SI trace $\phi \in A$ClockTrace, two transitions with labels ι, ι', and a time-tick trace segment ϕ_1 such that*

$$\begin{aligned}\phi = \phi' \xrightarrow{\iota''} \left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}\right), \mathtt{I} \xrightarrow{\iota} \phi_2 \xrightarrow{\iota'} \phi'' \\ \wedge\, \mathtt{TickAndSnapshot}\,(\phi_1, r) \wedge (\iota' = (cl, r, (\mathtt{P}, k, v)\,, n'', n) \vee \iota' = (cl, r, n, \mathtt{E})) \\ \wedge\, \exists cl', r', r'', l', k', v', n', n_1.\, cl \neq cl' \wedge r \neq r'' \wedge n_1 \geq n \\ \wedge\, (\iota = (cl', r', (l', k', v'), n', n_1) \vee \iota = (cl', r'', n', \mathtt{S}) \vee \iota = (cl', r', n', \mathtt{E}) \vee \iota = (r'', n'))\end{aligned}$$

where the rest free variables are universally quantified. The transition labelled by ι, together with the time-tick trace segment ϕ_1, can be moved to the left, that is, there exists a new time-tick trace segment ϕ_2, a new label ι_1 and a new equivalent trace such that

$$\mathtt{TickAndSnapshot}\,(\phi_2, r) \wedge \left(\phi' \xrightarrow{\iota''} \phi_2 \xrightarrow{\iota'} \_ \xrightarrow{\iota_1} \phi''\right) \simeq \phi. \tag{A.12}$$

Proof. Assume a trace $\phi = \phi' \xrightarrow{\iota''} \left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}\right), \mathtt{I} \xrightarrow{\iota} \phi^* \xrightarrow{\iota'} \phi''$. Assume $\iota' = (cl, r, (\mathtt{P}, k, v)\,, n'', n)$ or $\iota' = (cl, r, n, \mathtt{E})$. Let $\left(\left(\mathcal{R}', \tilde{\mathcal{U}}'\right), \mathcal{E}'\right) = \phi''_{|0}$ be the first state of the trace segment ϕ''. Let $(\tilde{\mathcal{K}}, n^{**}) = \mathcal{R}\,(r)$ and $\left(\tilde{\mathcal{K}}', n^{***}\right) = \mathcal{R}\,(r')$. By CSIPrepare and CSICommit,

$$\tilde{\mathcal{K}} \sqsubseteq \tilde{\mathcal{K}}' \wedge n^{**} < n^{***} \wedge \mathcal{R}' = \mathcal{R}\left[r \mapsto (\tilde{\mathcal{K}}', n^{***})\right] \wedge \tilde{\mathcal{U}} \sqsubseteq \tilde{\mathcal{U}}' \tag{A.32}$$

We perform case analysis on the label ι.

(1) [**Case**: $\iota = (cl', r', n', \mathtt{S})$] We immediately have $r \neq r'$ and $cl \neq cl'$. By CSISnapshot, the transition labelled by ι only depends on the state of r, that is, $\mathcal{R}\,(r') = (\_, n')$. This transition also does not change the state of the local time. By Eq. (A.32), the state of r remain unchanged in $\mathcal{R}'$, that is, $\mathcal{R}\,(r') = \mathcal{R}'(r')$. Therefore, there exists a new time-tick trace segment ϕ^{**} and a runtime program $\mathtt{I}''$ such that

$$\left(\phi' \xrightarrow{\iota''} \phi^{**} \xrightarrow{\iota'} \left(\mathcal{R}', \tilde{\mathcal{U}}', \mathcal{E}'\right), \mathtt{I}'' \xrightarrow{\iota} \phi''\right) \simeq \phi$$

where ϕ^{**} a trace contains the same steps as ϕ^*.

(2) [**Case**: $\iota = (cl', r', (\mathtt{R}, k', v'), n')$ **or** $\iota = (cl', r', (\mathtt{W}, k', v'), n')$] We immediately have $cl \neq cl'$. Note that any start label in the trace segment ϕ^* cannot come from the client cl', because the transaction corresponds to the label ι have not finish. Therefore, by Prop. A.21, we move ι to the right until we have the proof.

(3) [**Case**: $\iota = (cl', r', (\mathtt{P}, k', v'), n', n^*)$] We immediately have $cl \neq cl'$ and $n^* \geq n$. There are two cases: $r = r'$ or $r \neq r'$.

(i) [**Case**: $r = r'$] There are two cases: $\iota' = (cl, r, (\mathtt{P}, k, v), n'', n)$ or $\iota' = (cl, r, n, \mathtt{E})$.

(a) [**Case**: $\iota' = (cl, r, (\mathtt{P}, k, v), n'', n)$] Note that because both ι and ι' operate on the same shard r, and by CSIPREPARE that there must be no concurrent writes on the keys k and k' until the preparation time n'' and n' respectively (modelled by the premise $\forall i.\, 0 \leq i < n \Rightarrow \mathsf{TimeOf}\left(\tilde{\mathcal{V}}(i)\right) < n'$ in CSICOMMIT), then $k \neq k'$. These two preparation steps, ι and ι', also stop the preparation step from other transactions that want to write to keys k and k' in between n'' and n, and between n' and n^* respectively. Now consider the first preparation step ι. It is safe to advanced the preparation time of this transition up to the actual commit time n^*. Recall that: A. by the hypothesis, we have $n \leq n^*$; B. any preparation steps must happen before the actual commit, that is, $n'' \leq n$ and $n' \leq n^*$ respectively; C. for the same shard, the time must monotonically increase, thus $n' \leq n''$. Therefore, for the two preparation steps in the original trace, we have

$$n' \leq n'' \leq n \leq n^*.$$

This means it is allowed to delay the preparation step ι after ι', even thought the step ι might be assigned a larger time as n''. Let $\iota^* = (cl', r', (\mathtt{P}, k', v'), n'', n^*)$ Therefore, there exists a new trace segment ϕ^{**} such that:

$$\left(\phi' \xrightarrow{\iota''} \phi^{**} \xrightarrow{\iota'} \left(\mathcal{R}', \tilde{\mathcal{U}}', \mathcal{E}'\right), \mathtt{I}'' \xrightarrow{\iota^*} \phi''\right) \simeq \phi.$$

(b) [**Case**: $\iota' = (cl, r, n, \mathtt{E})$] Follow a similar argument as the previous case, we have

$$n' \leq n \leq n^*.$$

Let $\iota^* = (cl', r', (\mathtt{P}, k', v'), n'', n^*)$ Therefore, there exists a new trace segment ϕ^{**} such that:

$$\left(\phi' \xrightarrow{\iota''} \phi^{**} \xrightarrow{\iota'} \left(\mathcal{R}', \tilde{\mathcal{U}}', \mathcal{E}'\right), \mathtt{I}'' \xrightarrow{\iota^*} \phi''\right) \simeq \phi.$$

(ii) [**Case**: $r \neq r'$] Since steps from ϕ^* and ι operates on different shard, it is trivial to see that there exists a new time-tick trace segment ϕ^{**} and a runtime program $\mathtt{I}''$

such that

$$\left(\phi' \xrightarrow{\iota''} \phi^{**} \xrightarrow{\iota'} \left(\mathcal{R}', \tilde{\mathcal{U}}', \mathcal{E}'\right), \mathtt{I}'' \xrightarrow{\iota} \phi''\right) \simeq \phi. \qquad \blacksquare$$

Proposition A.22 (Clock-SI unique transactional identifiers)**.** *Assume an annotated normalised Clock-SI trace ϕ, and the kv-store trace induced by the Clock-SI trace η, defined by $\eta = \mathsf{ClockToKVTrace}(\phi)$. Let $(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}, \mathtt{I}) = \mathsf{Last}(\phi)$ and $(\mathcal{K}, \mathcal{U}, \mathcal{E}, \mathtt{P}) = \mathsf{Last}(\eta)$. The transaction identifiers in $\mathcal{K}$ are unique in that:*

$$\wedge\, \forall \phi' \in \phi.\, \forall cl, r.\, \left(\mathtt{CLOCKAtomic}\left(\phi', cl, r, n\right) \Rightarrow \tilde{t}_{cl}^{(n,\_)} \in \mathcal{K}\right) \tag{A.33}$$

$$\begin{aligned} &\wedge\, \forall \tilde{t}_{cl}^{(n,m)} \in \mathcal{K}.\, \exists \phi' \in \phi.\, \exists cl, r. \\ &\quad \left(\mathtt{CLOCKAtomic}\left(\phi', cl, r, n\right) \wedge \forall \phi'' \in \phi.\, \left(\mathtt{CLOCKAtomic}\left(\phi'', cl, r, n\right) \Rightarrow \phi' = \phi''\right)\right). \end{aligned} \tag{A.34}$$

Proof. By the definition of $\mathsf{ClockToKVTrace}$, it is easy to see that Eq. (A.33) and

$$\forall \tilde{t}_{cl}^{(n,m)} \in \mathcal{K}.\, \exists \phi' \in \phi.\, \exists cl, r.\, \mathtt{CLOCKAtomic}\left(\phi', cl, r, n\right).$$

We now prove that

$$\forall \tilde{t}_{cl}^{(n,m)} \in \mathcal{K}.\, \forall \phi', \phi'' \in \phi.\, \forall cl, r.\, \mathtt{CLOCKAtomic}\left(\phi', cl, r, n\right) \wedge \mathtt{CLOCKAtomic}\left(\phi'', cl, r, n\right) \Rightarrow \phi' = \phi''.$$

The above can be derived from the follows,

$$\begin{aligned} &\forall \phi', \phi'' \in \phi.\, \forall cl \in \mathrm{CID}.\, \forall r, r' \in \textsc{ClockShard}.\, \forall n, n' \in \mathbb{N}. \\ &\quad \mathtt{CLOCKAtomic}\left(\phi', cl, r, n\right) \wedge \mathtt{CLOCKAtomic}\left(\phi'', cl, r', n'\right) \\ &\qquad \phi = \cdots \Rightarrow \phi' \Rightarrow \cdots \Rightarrow \phi' \Rightarrow \cdots \Rightarrow n < n'. \end{aligned}$$

The above can be directly derived by Lemma A.23: the snapshot time must be strictly greater then the client local time, and the commit time must be greater or equal to the snapshot time. Therefore, Eq. (A.34) holds. $\blacksquare$

Lemma A.23 (Monotonic Clock-SI client local times)**.** *Client local times monotonically increases: given Clock-SI databases $\mathcal{R}, \mathcal{R}'$, local time environments $\tilde{\mathcal{U}}, \tilde{\mathcal{U}}'$, local stack environments $\mathcal{E}, \mathcal{E}'$, runtime programs $\mathtt{I}, \mathtt{I}'$, and label ι,*

$$\begin{aligned} \left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}\right), \mathtt{I} \xrightarrow{\iota} \left(\mathcal{R}', \tilde{\mathcal{U}}', \mathcal{E}'\right), \mathtt{I}' \wedge \forall cl \in \mathsf{Dom}\left(\tilde{\mathcal{U}}\right).\, \forall r \in \mathsf{Dom}\left(\mathcal{R}\right).\, \forall n.\, \tilde{\mathcal{U}}\left(cl\right) \leq \tilde{\mathcal{U}}'(cl) \\ \wedge \left(\iota = (cl, r, n, \mathtt{S}) \Rightarrow \tilde{\mathcal{U}}\left(cl\right) < \tilde{\mathcal{U}}'(cl).\right) \end{aligned}$$

Proof. We perform case analysis on the label ι.

(1) [**Case**: $\iota = (cl, r, n, \mathtt{S})$] By the premise of CSISnapshot, we have $\tilde{\mathcal{U}}(cl) < \tilde{\mathcal{U}}'(cl)$. Other clients cl' such that $cl' \neq cl$, remains unchanged, therefore $\tilde{\mathcal{U}}(cl') = \tilde{\mathcal{U}}'(cl')$.

(2) [**Case**: $\iota = (cl, r, (l, k, v), n)$ **with** $l \in \{\mathtt{W}, \mathtt{R}, \mathtt{P}\}$] By CSIWrite, CSIReadL, CSIReadS and CSIPrepare, the client local environment remains unchanged, therefore $\tilde{\mathcal{U}}(cl) = \tilde{\mathcal{U}}'(cl)$ for all clients cl.

(3) [**Case**: $\iota = (cl, r, n, \mathtt{E})$ **or** $\iota = (r, n)$] By CSICommit and CSITick, the client local environment remains unchanged, therefore $\tilde{\mathcal{U}}(cl) = \tilde{\mathcal{U}}'(cl)$ for all clients cl. ■

Proposition A.24 (Well-formed Clock-SI views)**.** *Assume an annotated normalised Clock-SI trace ϕ, and the kv-store trace induced by the Clock-SI trace $\eta = \mathsf{ClockToKVTrace}(\phi)$. Let $(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}, \mathtt{I}) = \mathsf{Last}(\phi)$ and $(\mathcal{K}, \mathcal{U}, \mathcal{E}, \mathtt{P}) = \mathsf{Last}(\eta)$. The views in $\mathcal{U}$ are well-formed:*

$$\forall cl \in \mathsf{Dom}(\mathcal{U}) \,.\, \mathtt{WfView}(\mathcal{K}, \mathcal{U}(cl)) \,.$$

Proof. By the definition of ClockTrace, all keys in all shards are initialised at time ZERO, which implies Eq. (4.5) in Def. 4.6. Assume a client cl and its view $u = \mathcal{U}(cl)$. By the definition of ClockView that u contains transactions committed before the client local time $\tilde{\mathcal{U}}(cl)$, it is easy to see that the view is in range, defined in Eq. (4.6) in Def. 4.6. Because new versions of a transaction must commit at the same time. This view includes all or none of the effect of transactions defined in Eq. (4.7) in Def. 4.6. ■

Proposition A.25 (Well-formed Clock-SI fingerprints)**.** *Assume an atomic trace segment ϕ such that: for some $\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}, \mathtt{I}, \mathtt{T}, \mathcal{B}, n, r, \mathtt{C}$,*

$$\mathtt{CLOCKAtomic}(\phi, cl, r) \wedge (\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}, \mathtt{I}) = \phi_{|0} \wedge \mathtt{I}(cl) = [\ \mathtt{T}\ ]^{\emptyset, \mathcal{B}}_{n,r} \,;\, \mathtt{C}. \tag{A.35}$$

Then, the fingerprint $\mathcal{F} = \mathsf{ClockFp}(\emptyset, \phi)$ satisfies the follows: for some $\mathcal{R}', \tilde{\mathcal{U}}', \mathcal{E}', \mathtt{I}', \mathtt{T}', \mathcal{B}', n, r, \mathtt{C}$,

$$(\mathcal{R}', \tilde{\mathcal{U}}', \mathcal{E}', \mathtt{I}') = \mathsf{Last}(\phi) \wedge \mathtt{I}(cl) = [\ \mathtt{T}'\ ]^{\mathcal{F}, \mathcal{B}'}_{n,r} \,;\, \mathtt{C}. \tag{A.36}$$

Proof. Fix the initial fingerprint $\mathcal{F}$, we prove by induction on the length of trace ϕ.

(1) [**Base Case**: $\phi = \emptyset$] By the definition, we have $\mathsf{ClockStaticProg}(\emptyset, \phi) = \emptyset$. It is trivial that $\emptyset$ is a well-formed fingerprint and Eq. (A.36) holds.

(2) [**Inductive Case**: $\phi = \phi' \xrightarrow{\iota} \left(\mathcal{R}', \tilde{\mathcal{U}}', \mathcal{E}'\right), \mathtt{I}'$] Given the client cl, assume the fingerprint $\mathcal{F}'$ such that

$$(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}, \mathtt{I}) = \mathsf{Last}(\phi') \wedge \mathtt{I}(cl) = [\ \mathtt{T}\ ]^{\mathcal{F}', \mathcal{B}}_{n,r} \,;\, \mathtt{C}.$$

and assume fingerprint $\mathcal{F}''$ such that

$$\mathtt{I}'(cl) = [\ \mathtt{T}'\ ]^{\mathcal{F}'', \mathcal{B}'}_{n,r} \,;\, \mathtt{C}$$

where the free variables are existentially quantified. By the inductive hypothesis, we immediately have $\mathsf{ClockStaticProg}(\mathcal{F}, \phi') = \mathcal{F}'$. We perform case analysis on the label ι.

(i) [**Case**: $\iota = (cl, r, (\mathtt{W}, k, v), n, n')$] By the rule CSIWRITE, we have $\mathcal{F}'' = \mathcal{F}' \lessdot (\mathtt{W}, k, v)$. By the definition of $\mathsf{ClockStaticProg}$, we have

$$\mathsf{ClockStaticProg}(\mathcal{F}, \phi) = \mathsf{ClockStaticProg}(\mathcal{F}, \phi') \lessdot (\mathtt{W}, k, v) = \mathcal{F}' \lessdot (\mathtt{W}, k, v) = \mathcal{F}''$$

which implies Eq. (A.36).

(ii) [**Case**: $\iota = (cl, r, (\mathtt{R}, k, v), n, n')$] Consider if $\mathcal{F}'$ contains an entry for the key k, which corresponds to the two rules CSIREADL and CSIREADS.

(a) [**Case**: **CSIReadL**] In this case, we have $(l, k, v) \in \mathcal{F}'$ for some $l \in \{\mathtt{W}, \mathtt{R}\}$ and value v. By the premise of CSIREADL, we have $\mathcal{F}' = \mathcal{F}''$. By the definition of $\mathsf{ClockStaticProg}$, we have

$$\mathsf{ClockStaticProg}(\mathcal{F}, \phi) = \mathsf{ClockStaticProg}(\mathcal{F}, \phi') \lessdot (\mathtt{R}, k, v) = \mathcal{F}' \lessdot (\mathtt{R}, k, v) = \mathcal{F}''$$

which implies Eq. (A.36).

(b) [**Case**: **CSIReadS**] In this case, we have $(l, k, v) \notin \mathcal{F}'$ for all $l \in \{\mathtt{W}, \mathtt{R}\}$ and value v. By the premise of CSIREADL, we have $\mathcal{F}'' = \mathcal{F}' \lessdot (\mathtt{R}, k, v)$. By the definition of $\mathsf{ClockStaticProg}$, we have

$$\mathsf{ClockStaticProg}(\mathcal{F}, \phi) = \mathsf{ClockStaticProg}(\mathcal{F}, \phi') \lessdot (\mathtt{R}, k, v) = \mathcal{F}' \lessdot (\mathtt{R}, k, v) = \mathcal{F}''$$

which implies Eq. (A.36). ■

Theorem 6.39 (Well-formed Clock-SI centralised kv-store)**.** *Assume an annotate normalised Clock-SI trace ϕ and the kv-store trace induced by the Clock-SI trace $\eta = \mathsf{ClockToKVTrace}(\phi)$. Let $(\mathcal{K}, \mathcal{U}, \mathcal{E}, \mathtt{P}) = \mathsf{Last}(\eta)$ and $(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}, \mathtt{I}) = \mathsf{Last}(\phi)$. The final kv-store $\mathcal{K}$ is well-formed, and $\mathcal{K}$ only contains versions that exist in $\mathcal{R}$, and vice versa, that is,*

$$\forall k, i, v, cl, n, m, T.\, \mathcal{K}(k, i) = (v, \tilde{t}_{cl}^{(n,m)}, T) \Rightarrow \exists r, \tilde{\mathcal{K}}, i'.$$
$$r = \mathsf{ShardOf}_{\mathcal{R}}(k) \wedge (\tilde{\mathcal{K}}, \_) = \mathcal{R}(r) \wedge \tilde{\mathcal{K}}(k, i') = (v, n, \_), \tag{A.13}$$
$$\forall k, r, \tilde{\mathcal{K}}, i, v, n.\, r = \mathsf{ShardOf}_{\mathcal{R}}(k) \wedge (\tilde{\mathcal{K}}, \_) = \mathcal{R}(r) \wedge \tilde{\mathcal{K}}(k, i) = (v, n, \_) \Rightarrow$$
$$\exists i', cl, m, T.\, \mathcal{K}(k, i) = (v, \tilde{t}_{cl}^{(n,m)}, T). \tag{A.14}$$

Proof. We prove by induction on Clock-SI trace ϕ.

(1) [**Base Case**: $\phi = \left(\mathcal{R}_0, \tilde{\mathcal{E}}_0, \mathcal{E}_0\right), \mathtt{P}_0$] By the definition of $\mathsf{ClockToKVTrace}$, we know that $\eta = (\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}_0), \mathtt{P}_0$. It is trivial that the initial kv-store $\mathcal{K}_0$ is well-formed, and $\mathcal{R}_0$ and $\mathcal{K}_0$ satisfy Eqs. (6.13) and (6.14).

(2) [**Inductive Case**: $\phi = \phi' \xrightarrow{\iota} \phi'' \xrightarrow{\iota'} \left(\mathcal{R}'', \tilde{\mathcal{U}}'', \mathcal{E}''\right), \mathtt{I}''$] Let $(\mathcal{R}', \tilde{\mathcal{U}}', \mathcal{E}', \mathtt{I}) = \mathsf{Last}(\phi')$ and $\eta' = \mathsf{ClockToKVTrace}(\phi')$. By inductive hypothesis, the final kv-store $\mathcal{K}'$ such that $(\mathcal{K}', \mathcal{U}', \_, \_) = \mathsf{Last}(\eta')$, is well-formed, and $\mathcal{R}'$ and $\mathcal{K}'$ satisfy Eqs. (6.13) and (6.14). Now let consider the next step: there are two possible cases for ϕ''.

(i) [**Cases**: $\iota = (cl, r, n, \mathtt{S})$ **or** $\iota = (r, n)$**, and both with** $\phi'' = \emptyset$] By the definition of $\mathsf{ClockToKVTrace}$, $\mathsf{ClockToKVTrace}(\phi) = \mathsf{ClockToKVTrace}(\phi')$, and by inductive hypothesis, we have the proof.

(ii) [**Cases**: $\iota = (cl, r, (l, k, v), n, n)$] Let $\phi^* = \mathsf{Last}(\phi) \xrightarrow{\iota} \phi' \xrightarrow{\iota'} \left(\mathcal{R}'', \tilde{\mathcal{U}}'', \mathcal{E}''\right), \mathtt{I}''$. In this case, we have $\mathtt{CLOCKAtomic}(\phi^*, cl, r, n)$. Let $u = \mathsf{ClockView}(\mathcal{K}', n)$ be the view and $\mathcal{F} = \mathsf{ClockFp}(\emptyset, \phi^*)$ be the fingerprint. By the definition of $\eta = \mathsf{ClockToKVTrace}(\phi)$, we have

$$\eta = \eta' \xrightarrow{(cl,u,\mathcal{F})}_{\mathtt{ET}} \left(\mathcal{K}'', \mathcal{U}'\left[cl \mapsto u'\right], \mathcal{E}''\right), \mathsf{ClockStaticProg}(\mathtt{I})$$

where $\mathcal{K}'' = \mathsf{UpdateKV}\left(\mathcal{K}, u, \mathcal{F}, \tilde{t}_{cl}^{(n,m)}\right)$ and $u = \mathsf{ClockView}(\mathcal{K}'', n)$. Note that we have the following results:

(a) by inductive hypothesis, the kv-store $\mathcal{K}''$ is well-formed;

(b) by Prop. A.24, the pre-view u is well-formed on $\mathcal{K}''$;

(c) by Prop. A.25, the fingerprint $\mathcal{F}$ is well-formed fingerprint; and

(d) by Prop. A.22, the transaction identifier $\tilde{t}_{cl}^{(n,m)}$ must be a unique transaction identifier and by Lemma A.23, $\tilde{t}_{cl}^{(n,m)} \in \mathsf{NextTxID}(\mathcal{K}, cl)$.

Given above, by Theorem 4.21, we have that $\mathcal{K}'' = \mathsf{UpdateKV}\left(\mathcal{K}, u, \mathcal{F}, \tilde{t}_{cl}^{(n,m)}\right)$ is well-formed.

Now we prove Eqs. (6.13) and (6.14). Note that $\mathcal{K}'$ and $\mathcal{R}'$ satisfy Eqs. (6.13) and (6.14), Consider the new versions in $\mathcal{K}''$. Take a write operation such that $(\mathtt{W}, k, v) \in \mathcal{F}$. This means that, if and only if, $\mathcal{K}''(k, |\mathcal{K}''(k)| - 1) = (v, \tilde{t}_{cl}^{(n,m)}, \emptyset)$. By definition of $\mathcal{F} = \mathsf{ClockFp}(\emptyset, \phi^*)$, if and only if, there exists two transitions, $(cl, r, (\mathtt{W}, k, v), n, n)$ and $(cl, r, (\mathtt{W}, k, v), n', n)$ for some preparation time n' such $n' > n$, in the trace ϕ^*. By the definition of CSIPrepare and CSICommit, if and only if, we have $\tilde{\mathcal{K}}''(k, \left|\tilde{\mathcal{K}}''(k)\right| - 1) = (v, n, \mathtt{committed})$ for $(\tilde{\mathcal{K}}'', \_) = \mathcal{R}''(\mathsf{ShardOf}_{\mathcal{R}}(k))$. Thus, Eqs. (6.13) and (6.14) hold. ■

Theorem 6.40 (Clock-SI traces satisfying snapshot isolation)**.** *Given an annotated normalise Clock-SI trace ϕ, the kv-store trace induced by the Clock-SI trace $\eta = \mathsf{ClockToKVTrace}(\phi)$ is a trace that can be obtained with the execution test for* `SI`.

Proof. Given a normalised Clock-SI trace ϕ, by Theorem 6.39, the induced kv-store trace $\eta = \mathsf{ClockToKVTrace}(\phi)$ is a trace that can be obtained under $\mathtt{ET}_{\top}$. We prove that every transition in the trace η satisfies the constrains of $\mathtt{ET}_{\mathtt{PSI}}$ by induction on ϕ.

(1) [**Base Case**: $\phi = \left(\mathcal{R}_0, \tilde{\mathcal{E}}_0, \mathcal{E}_0\right), \mathtt{P}_0$] By the definition of $\mathsf{ClockToKVTrace}$, we know that $\eta = (\mathcal{K}_0, \mathcal{U}_0, \mathcal{E}_0), \mathtt{P}_0$, and there is nothing to prove since no transition has been made.

(2) [**Inductive Case**: $\phi = \phi' \xrightarrow{\iota} \phi'' \xrightarrow{\iota'} \left(\mathcal{R}'', \tilde{\mathcal{U}}'', \mathcal{E}''\right), \mathtt{I}''$] If $\iota = (r, n)$ or $\iota = (cl, r, n, \mathtt{S})$, by the definition of $\mathsf{ClockToKVTrace}$, we have $\mathsf{ClockToKVTrace}(\phi) = \mathsf{ClockToKVTrace}(\phi')$, and thus by inductive hypothesis, we have the proof. Now consider

$$\begin{aligned}\phi = \phi' \xrightarrow{\iota} \phi'' \xrightarrow{\iota'} \left(\mathcal{R}'', \tilde{\mathcal{U}}'', \mathcal{E}''\right), \mathtt{I}'' \\ \wedge \iota = (cl, r, (l, k, v), m, n) \wedge l \in \{\mathtt{R}, \mathtt{W}\} \wedge \iota' = (cl, r, n, \mathtt{E})\end{aligned}$$

Let $\phi^* = \mathsf{Last}(\phi) \xrightarrow{\iota} \phi' \xrightarrow{\iota'} \left(\mathcal{R}'', \tilde{\mathcal{U}}'', \mathcal{E}''\right), \mathtt{I}''$. Let $\eta' = \mathsf{ClockToKVTrace}(\phi')$ and $(\mathcal{K}', \mathcal{U}', \_, \_) = \mathsf{Last}(\eta')$. By the definition of $\mathsf{ClockToKVTrace}$, we have

$$\mathsf{ClockToKVTrace}(\phi) = \eta' \xrightarrow{(cl,u,\mathcal{F})}_{\top} (\mathcal{K}'', \mathcal{U}'[cl \mapsto u'], \mathcal{E}''), \mathtt{P}'',$$

where

$$\begin{aligned}\mathcal{F} = \mathsf{ClockFp}(\emptyset, \phi^*) \wedge u = \mathsf{ClockView}(\mathcal{K}', n) \\ \wedge \mathcal{K}'' = \mathsf{UpdateKV}\left(\mathcal{K}', u, \mathcal{F}, \tilde{t}_{cl}^{(n,m)}\right) \wedge u' = \mathsf{ClockView}(\mathcal{K}'', n) \wedge \mathtt{P}' = \mathsf{ClockStaticProg}(\mathtt{I}).\end{aligned}$$

Note that in the proof we call n as the snapshot time and n as the commit time. By the definition of PSI, we need to prove the following results:

$$u \sqsubseteq u' \tag{A.37}$$

$$\forall t \in \mathcal{K}'' \setminus \mathcal{K}'. \forall k \in \textsc{Key}. \forall i \in \mathbb{N}. \mathsf{WriterOf}(\mathcal{K}''(k, i)) \xrightarrow{\mathsf{SO}^?} t \Rightarrow i \in u'(k) \tag{A.38}$$

$$\mathtt{PreClosed}\left(\mathcal{K}', u, \left((\mathsf{WR}_{\mathcal{K}'} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}'})^+; \mathsf{RW}_{\mathcal{K}'}{}^?\right)^*\right) \tag{A.39}$$

$$\mathtt{PreClosed}\left(\mathcal{K}, u, \bigcup_{(\mathtt{W}, k, \_) \in \mathcal{F}} \mathsf{WW}_{\mathcal{K}'}^{-1}(k)\right) \tag{A.40}$$

We prove Eqs. (A.37) to (A.40) separately.

(i) [**Case**: **Eq. (A.37)**] By Lemma A.23, we have the commit time n is greater or equal to the snapshot time m. This means that $m \leq n$ and thus $u = \mathsf{ClockView}(\mathcal{K}', m) \sqsubseteq \mathsf{ClockView}(\mathcal{K}'', n) = u'$.

(ii) [**Case**: **Eq. (A.38)**] Consider any write operation $(\mathtt{W}, k, v) \in \mathcal{F}$. By CSIPrepare, CSICommit and the definition of annotated Clock-SI traces, AClockTrace, there must exist a write step of the form $(cl, r, (\mathtt{W}, k, v), m, n)$ in the Clock-SI trace segment ϕ' such that $m \leq n$. Recall that $\mathcal{K}'' = \mathsf{UpdateKV}(\mathcal{K}', u, \mathcal{F}, t_{cl}^{n,m})$, thus we know that

$$\mathcal{K}''(k, |\mathcal{K}'(k)| - 1) = (v, t_{cl}^{n,m}, \emptyset).$$

By the definition of $u' = \mathsf{ClockView}\,(\mathcal{K}'', n)$, we have $(|\mathcal{K}''(k)| - 1) \in u'(k)$.

(iii) [**Case**: **Eq. (A.39)**] We prove the correspondence between the four relations, WR, WW, RW and SO, with respect to the snapshot and commit times. Let consider two transactions $t_{cl'}^{n',m'}, t_{cl''}^{n'',m''}$ that accessed key key in $\mathcal{K}'$. Note that $t_{cl'}^{n',m'}, t_{cl''}^{n'',m''}$ might be the writers or the readers of some versions. By Theorem 6.39, there must exist read or write steps that corresponds to $t_{cl'}^{n',m'}, t_{cl''}^{n'',m''}$ in the trace ϕ'. Without losing generality, we assume the steps for $t_{cl'}^{n',m'}$ appear before the steps for $t_{cl''}^{n'',m''}$:

$$\phi' = \cdots \xrightarrow{(cl,r,(l',k,v'),m',n')} \cdots \xrightarrow{(cl,r,(l'',k,v''),m'',n'')} \cdots \wedge l', l'' \in \{\mathtt{W}, \mathtt{R}\}$$

Given that the snapshot time m of the new transaction $t_{cl}^{n,m}$, assume that $n'' < m$. This means that $t_{cl}^{n,m}$ observes $t_{cl''}^{n'',m''}$. Now we prove that $t_{cl}^{n,m}$ also observes $t_{cl'}^{n',m'}$ for the following cases.

(a) [**Case**: $(t_{cl'}^{n',m'}, t_{cl''}^{n'',m''}) \in \mathsf{WR}_{\mathcal{K}'}\,(k)$] This means cl'' read a version written by cl': there exists an index i such that

$$t_{cl'}^{n',m'} = \mathsf{WriterOf}\,(\mathcal{K}'(k,i)) \wedge t_{cl''}^{n'',m''} \in \mathsf{ReadersOf}\,(\mathcal{K}'(k,i))\,.$$

We then have $l' = \mathtt{W}$ and $'' = \mathtt{R}$. By Lemma A.26 cl'' must have a snapshot time m'' greater or equal to the commit time n', and then by Lemma A.23, we have $n' < n''$. In other words,

$$(t_{cl'}^{n',m'}, t_{cl''}^{n'',m''}) \in \mathsf{WR}_{\mathcal{K}'} \Rightarrow n' < m'' \leq n''. \tag{A.41}$$

(b) [**Case**: $(t_{cl'}^{n',m'}, t_{cl''}^{n'',m''}) \in \mathsf{WW}_{\mathcal{K}'}\,(k)$] This means cl'' write a new version for k after the version written by cl':

$$t_{cl'}^{n',m'} = \mathsf{WriterOf}\,(\mathcal{K}'(k,i)) \wedge t_{cl''}^{n'',m''} \in \mathsf{WriterOf}\,(\mathcal{K}'(k,i'))\; i < i'$$

The transaction $t_{cl''}^{n'',m''}$ commits after $t_{cl'}^{n',m'}$ in the kv-store trace η', thus in the Clock-SI trace ϕ'. By CSIPrepare rule, when cl'' prepares the new version, there must be no version that commit after the snapshot time of cl'', which means

$$(t_{cl'}^{n',m'}, t_{cl''}^{n'',m''}) \in \mathsf{WW}_{\mathcal{K}} \Rightarrow n' < m'' \leq n''. \tag{A.42}$$

(c) [**Case**: $(t_{cl'}^{n',m'}, t_{cl'}^{n'',m''}) \in \mathsf{SO}$] Because the local times for Clock-SI clients monotonically increase, we know that

$$(t_{cl'}^{n',m'}, t_{cl'}^{n'',m''}) \in \mathsf{SO} \Rightarrow n' < m'' \leq n''. \tag{A.43}$$

(d) [**Case**: $t_{cl'}^{n',m'} \xrightarrow{R} t_{cl^*}^{n^*,m^*} \xrightarrow{\mathsf{RW}_{\mathcal{K}'}(k)} t_{cl''}^{n'',m''}$ **with** $R \in \{\mathsf{WR}_{\mathcal{K}'}, \mathsf{WW}_{\mathcal{K}'}, \mathsf{SO}\}$]
This means there exist two indices i, i' such that

$$i < i' \wedge \exists T.\, \mathcal{K}\,(k,i) = (\_,\_,T) \wedge t_{cl^*}^{n^*} \in T \wedge \mathcal{K}\,(k,i') = (\_, t_{cl''}^{n'',m''}, \_)$$

Consider the transaction $t_{cl^*}^{n^*,m^*}$ in the Clock-SI trace. By Lemma A.26, The snapshot time m^* must be strictly smaller than than n'' (otherwise it contradict to Lemma A.26), that is, $m^* < n''$. By $t_{cl'}^{n',m'} \xrightarrow{R} t_{cl^*}^{n^*,m^*}$ for $R \in \{\mathsf{WR}_{\mathcal{K}'}, \mathsf{WW}_{\mathcal{K}'}, \mathsf{SO}\}$ and Eqs. (A.41) to (A.43), we then have $n' < m^* < n^*$, that is,

$$t_{cl'}^{n',m'} \xrightarrow{R} t_{cl^*}^{n^*,m^*} \xrightarrow{\mathsf{RW}_{\mathcal{K}'}(k)} t_{cl''}^{n'',m''} \Rightarrow n' < n'' \tag{A.44}$$

Combine Eqs. (A.41) to (A.44), we have

$$(t_{cl'}^{n',m'}, t_{cl''}^{n'',m''}) \in \left((\mathsf{WR}_{\mathcal{K}} \cup \mathsf{SO} \cup \mathsf{WW}_{\mathcal{K}})^+; (\mathsf{RW}_{\mathcal{K}'}^?)\right)^* \Rightarrow n' < n''. \tag{A.45}$$

This means, by the definition of $u = \mathsf{ClockView}\,(\mathcal{K}, n)$, if the view u includes version written by $t_{cl''}^{n'',m''}$, then the view also includes version written by $t_{cl'}^{n',m'}$, thus we prove Eq. (A.39).

(iv) [**Case**: **Eq. (A.40)**] Recall that the new transaction $t_{cl}^{n,m}$. Consider any write operation $(\mathtt{W}, k, v) \in \mathcal{F}$. Let $r = \mathsf{ShardOf}_{\mathcal{R}}\,(k)$ be the shard that contains k, and $(\tilde{\mathcal{K}}, \_) = \mathcal{R}\,(r)$ be the local key-value store. By CSIPrepare, there must be no version for the key k with the committed or preparation time greater than m, that is,

$$\forall i, n'.\, \tilde{\mathcal{K}}\,(k,i) = (\_, n', \_) \Rightarrow n' < m.$$

By the definition of $u = \mathsf{ClockView}\,(\mathcal{K}, n)$, we prove Eq. (A.40). ■

Lemma A.26 (Clock-SI WR)**.** *Assume an annotated normalised Clock-SI trace ϕ, and the kv-store trace induced by the Clock-SI trace $\eta = \mathsf{ClockToKVTrace}\,(\phi)$. Given final kv-store $\mathcal{K}$ in η, that is, $(\mathcal{K}, \_, \_, \_) = \mathsf{Last}\,(\eta)$, and a write-read edge, $\left(\tilde{t}_{cl}^{(n,m)}, \tilde{t}_{cl'}^{(n',m')}\right) \in \mathsf{WR}_{\mathcal{K}}\,(k)$, then $\tilde{t}_{cl}^{(n,m)}$ is the latest transaction that write to key k and commits before $t_{cl'}^{n',m'}$ starts, that is,*

$$n = \mathsf{Max}\left(\left\{n'' \;\middle|\; \exists i, cl'', m''.\, \tilde{t}_{cl''}^{(n'',m'')} = \mathsf{WriterOf}\,(\mathcal{K}\,(k,i))\right\}\right) \wedge n < m'.$$

Proof. By the definition of $\mathsf{ClockToKVTrace}$ and $\left(\tilde{t}_{cl}^{(n,m)}, \tilde{t}_{cl'}^{(n',m')}\right) \in \mathsf{WR}_{\mathcal{K}}\,(k)$, the transaction $\tilde{t}_{cl'}^{(n',m')}$ commits after the commit of $\tilde{t}_{cl}^{(n,m)}$, that is, $m < m'$. This means that there exists a prefix of ϕ and a corresponding prefix of η, in which the last transaction is $\tilde{t}_{cl'}^{(n',m')}$. Let ϕ' denote the prefix of ϕ, and η' denote the prefix of η. We have $\eta' = \mathsf{ClockToKVTrace}\,(\phi')$. Since ϕ' is a

normalised trace, then

$$\phi' = \phi'' \xrightarrow{\iota} \phi^* \wedge \texttt{CLOCKAtomic}\left(\phi^*, cl', r, n'\right).$$

Let $\left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}, \texttt{I}\right) = \phi^*_{\restriction 0}$. Let $\eta'' = \mathsf{ClockToKVTrace}\left(\phi'' \xrightarrow{\iota} \left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}\right), \texttt{I}\right)$ and $\left(\mathcal{K}'', \mathcal{U}'', \mathcal{E}'', \texttt{P}''\right) = \mathsf{Last}\left(\eta''\right)$. By the definition of $\mathsf{ClockToKVTrace}$, we have

$$\eta' = \eta'' \xrightarrow{(cl,u,\mathcal{F})}_{\texttt{ET}} \left(\mathcal{K}^*, \mathcal{U}''\left[cl' \mapsto u'\right], \mathcal{E}^*\right), \texttt{P}^*$$

where

$$\begin{aligned} u = \mathsf{ClockView}\left(\mathcal{K}'', m'\right) \wedge \mathcal{F} = \mathsf{ClockFp}\left(\emptyset, \phi^*\right) \\ \wedge \mathcal{K}^* = \mathsf{UpdateKV}\left(\mathcal{K}'', u, \mathcal{F}, \tilde{t}_{cl'}^{(n',m')}\right) \wedge u' = \mathsf{ClockView}\left(\mathcal{K}^*, m'\right) \end{aligned}$$

Because $\tilde{t}_{cl'}^{(n',m')}$ read the key k in $\mathcal{K}^*$, then $(\texttt{R}, k, v) \in \mathcal{F}$. By the definition of $\mathcal{F} = \mathsf{ClockFp}\left(\emptyset, \phi^*\right)$ and Prop. A.25, there exists a read step that read from the shard r in ϕ^*:

$$\begin{aligned} \phi^* = \cdots \Rightarrow \left(\mathcal{R}, \_, \_\right), \texttt{I}\left[cl' \mapsto \left[\ \texttt{x} := \texttt{rd}\left(\texttt{E}\right); \texttt{T}\ \right]_{m',r}^{\mathcal{F}',\emptyset}\right] \xrightarrow{(cl',r,(\texttt{R},k,v),m',n')} \\ \left(\mathcal{R}, \_, \_\right), \texttt{I}\left[cl' \mapsto \left[\ \texttt{T}\ \right]_{m',r}^{\mathcal{F}' \lessdot (\texttt{R},k,v),\emptyset}\right] \Rightarrow \cdots \end{aligned}$$

where the free variables are existentially quantified. By the CSIReadS rule, we know that step $\left(cl', r, \left(\texttt{R}, k, v\right), m', n'\right)$ read the version of k with time n such that

$$\exists \tilde{\mathcal{K}}. \left(\tilde{\mathcal{K}}, \_\right) = \mathcal{R}\left(r\right) \wedge n = \mathsf{Max}\left(\left\{n' \;\middle|\; \exists i. \tilde{\mathcal{K}}\left(k, i\right) = \left(\_, n'\right) \wedge n' < m'\right\}\right). \tag{A.46}$$

Note that the state of $\mathcal{R}$ has not changed until the read step, since $\texttt{CLOCKAtomic}\left(\phi^*, cl', r, n'\right)$. Thus, by the definition of $\eta'' = \mathsf{ClockToKVTrace}\left(\phi'' \xrightarrow{\iota} \left(\mathcal{R}, \tilde{\mathcal{U}}, \mathcal{E}\right), \texttt{I}\right)$ and $\left(\mathcal{K}'', \mathcal{U}'', \mathcal{E}'', \texttt{P}''\right) = \mathsf{Last}\left(\eta''\right)$, we have $\mathcal{K}''(k, i) = (v, \tilde{t}_{cl}^{(n,m)}, \_)$ for some i, By the definition of $u = \mathsf{ClockView}\left(\mathcal{K}'', m'\right)$, we have $i \in u\left(k\right)$. Then by Eq. (A.46), and the definition of $\mathcal{K}^* = \mathsf{UpdateKV}\left(\mathcal{K}'', u, \mathcal{F}, \tilde{t}_{cl'}^{(n',m')}\right)$, we have the proof. ■