Reasoning about Weak Isolation Levels in Separation Logic

ANDERS ALNOR MATHIASEN, Aarhus University, Denmark
LÉON GONDELMAN, Aalborg University, Denmark
LÉON DUCRUET, Aarhus University, Denmark
AMIN TIMANY, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark

Consistency guarantees among concurrently executing transactions in local- and distributed systems, commonly referred to as isolation levels, have been formalized in a number of models. Thus far, no model can reason about executable implementations of databases or local transaction libraries providing weak isolation levels. Weak isolation levels are characterized by being highly concurrent and, unlike their stronger counterpart serializability, they are not equivalent to the consistency guarantees provided by a transaction library implemented using a global lock. Industrial-strength databases almost exclusively implement weak isolation levels as their default level. This calls for formalism as numerous bugs violating isolation have been detected in these databases.

In this paper, we formalize three weak isolation levels in separation logic, namely *read uncommitted*, *read committed*, and *snapshot isolation*. We define modular separation logic specifications that are independent of the underlying transaction library implementation. Historically, isolation levels have been specified using examples of executions between concurrent transactions that are not allowed to occur, and we demonstrate that our specifications correctly prohibit such examples. To show that our specifications are realizable, we formally verify that an executable implementation of a key-value database running the multi-version concurrency control algorithm from the original snapshot isolation paper satisfies our specification of snapshot isolation. Moreover, we prove implications between the specifications—snapshot isolation implies read committed and read committed implies read uncommitted—and thus the verification effort of the database serves as proof that all of our specifications are realizable. All results are mechanized in the Rocq proof assistant on top of the Iris separation logic framework.

CCS Concepts: • **Theory of computation → Logic and verification**; **Separation logic**; **Database theory**; *Distributed algorithms*; **Higher order logic**; **Hoare logic**; **Programming logic**; **Abstraction**; **Program specifications**; **Program verification**.

Additional Key Words and Phrases: Transactions, isolation levels, transactional memory, distributed systems, concurrency, formal verification, Iris, Rocq

ACM Reference Format:
Anders Alnor Mathiasen, Léon Gondelman, Léon Ducruet, Amin Timany, and Lars Birkedal. 2025. Reasoning about Weak Isolation Levels in Separation Logic. *Proc. ACM Program. Lang.* 9, ICFP, Article 246 (August 2025), 35 pages. https://doi.org/10.1145/3747515

Authors' Contact Information: Anders Alnor Mathiasen, Aarhus University, Aarhus, Denmark, alnor@cs.au.dk; Léon Gondelman, Aalborg University, Aalborg, Denmark, lego@cs.aau.dk; Léon Ducruet, Aarhus University, Aarhus, Denmark, leon.ducruet@ens-lyon.fr; Amin Timany, Aarhus University, Aarhus, Denmark, timany@cs.au.dk; Lars Birkedal, Aarhus University, Aarhus, Denmark, birkedal@cs.au.dk.

ACM 2475-1421/2025/8-ART246
https://doi.org/10.1145/3747515

1 Introduction

Transactions, defined as a set of operations considered as a single unit, is a common concept implemented by most modern databases and can also be encountered in local transactional memory systems. There exists multiple isolation levels for describing the allowed interaction between concurrently executing transactions. The strong isolation level of serializability [Papadimitriou 1979] often serves as an introduction to isolation levels. This level has consistency guarantees equivalent to what one would achieve in a transactional library protected by a global lock acquired at start time and released at commit time by the transactions. While serializability can be a useful property, it is often too strong for the desired use case and will negatively impact the throughput of the system. In contrast to serializability, weaker isolation levels allow for more concurrency and higher throughput. In fact, most modern commercial database systems, e.g., MySQL, Microsoft SQL, Oracle and PostgreSQL, have a weaker isolation level than serializability as default (Microsoft SQL, Oracle and PostgreSQL uses read committed [MicrosoftSQL 2025; Oracle 2025; PostgeSQL 2025] which we specify in this paper). With increased concurrency comes weaker consistency guarantees that by nature are harder to capture and reason about. As numerous bugs have been found in commercial database implementations violating isolation [Jepsen 2025], it is highly important to formally verify database implementations and their clients.

Existing Formalizations of Isolation Levels. A number of different formalizations have been developed to capture and reason about isolation levels. Historically, in the ANSI SQL standard of 1992, isolation levels were defined using examples of executions between concurrently executing transactions, so-called phenomena, which must be prohibited by the particular isolation level. The isolation levels of read uncommitted, read committed, repeatable read and serializability were defined using a phenomenon for each level except read uncommitted. This was later critiqued for being imprecise, which led to the invention of snapshot isolation and improved phenomena by Berenson et al. [1995]. With the phenomena as a basis, a number of different models have been developed to reason about isolation levels. Predominantly, models have been based on dependency graphs [Adya 1999; Adya et al. 2000], operational semantics [Cerone et al. 2015; Crooks et al. 2017; Kaki et al. 2017; Xiong 2020] or abstract executions [Burckhardt et al. 2012; Ketsman et al. 2023]. While the mentioned models can be used to define the desired consistency guarantees for transactions, in the form of isolation levels, they can not be applied to verify executable implementations of databases, transactional libraries and clients thereof. Recently, the separation logic framework Iris [Jung et al. 2018] has been used in Chang et al. [2023] to give a modular specification for serializability in the context of a key-value store. Separation logic is well known for its ability to specify various properties, using Hoare triples, and being able to reason about executable implementations [Gondelman et al. 2021; Nieto et al. 2023, 2022; Sharma et al. 2023]. The specification for serializability in [Chang et al. 2023] leverages that serializability is a strong property: A concise specification is given for a run operation that bundles together the start operation, commit operation and intermediate operations of a transaction into a single specification — something which is unfeasible for weaker isolation levels (this is discussed in Section 8). The argument of using separation logic to capture isolation levels and reason about executable code has yet to be completed by showing that separation logic can capture weak isolation levels.

Specifying Weak Isolation Levels in Separation Logic. The main contributions of this paper are formal separation logic specifications of weak isolation levels: read uncommitted, read committed and snapshot isolation. Together with serializability and repeatable read, these levels constitute the common levels found in commercial systems (repeatable read is identical to serializability except for its behavior regarding the predicate-read operation [Adya et al. 2000] which we do not model). To demonstrate that our specifications capture the desired isolation levels, we have, for the phenomena in the literature, created and proven examples of client transactions running against libraries adhering to our specifications which show that the phenomena are prohibited. We have chosen to verify the phenomena from the literature as clients because (1) They are concise examples that exercises edge cases, which a larger client implementation may not necessarily cover; and (2) these client-examples are used as the only specification in the SQL standard, and this is what industry databases use as specification. In addition to the phenomena, we have also verified a bank transfer example that represented a common use case for transactions (Section 5). Furthermore, we prove implications between our specifications, snapshot isolation implies read committed and read committed implies read uncommitted, which is consistent with the literature [Adya 1999]. Implications between separation logic specifications are unusual, as they establish that any implementation of the assumed specification also implements the implied specification.

Realizing Specifications. To show that our specifications are realizable by concrete implementations, we implement a database and verify that it meets our snapshot isolation specification. Together with the implication proofs between the specifications, this serve as a proof that all of our specifications are realizable. In more detail, we have used the distributed separation logic of Aneris [Krogh-Jespersen et al. 2020] to formally verify an in-memory single-node key-value store with support for transactions. In fact, we have implemented and verified the algorithm from the original snapshot isolation paper by Berenson et al. [1995]. The implementation uses multi-version concurrency control where for each key all previous updates are stored and ordered by timestamps. In addition to the operations in the API of the database, we have implemented utility functionalities on top of these operations in the form of wait and run operations. The wait operation returns once it has observed the key-value pair it has been given as argument, while the run operation wraps start and commit operations around a transaction body as in Chang et al. [2023]. We verify specifications for the wait operation for all the isolation levels we are considering, while we argue that only snapshot isolation is strong enough to have a concise specification for the run operation. Our implementations are written in OCaml and transpiled to AnerisLang for mechanized verification in the Rocq proof assistant.[1] In fact, all the results in this paper have been mechanized in Rocq. Aneris is an instantiation of the Iris separation logic framework with an OCaml like language (AnerisLang) and semantics for unreliable network communication. We emphasize that while we have used Aneris for the verification and implementation of a database, our specifications are independent of Aneris and can be used with other instantiations of Iris/separation logics.

Challenges. One of the key challenges of this work is to capture the isolation levels by modular separation logic specifications, with formal specifications for each of the individual operations on the key-value store — traditionally, isolation levels have not been formulated at this level of granularity with separate specifications for each of the operations: Existing transactional models either have a global view of all transactions, or are formulated with a high-level idealized operational semantics far from the semantics used to model the programming language on which we build in this paper. We emphasize that one of the advantages of our approach, using separation logic specifications, is modularity in because it enables one to reason formally about client programs that (possibly among other libraries) make use of the key-value store as a library. Another key challenge is to prove the implications between the specifications of the three isolation levels. The specifications are highly modular as they keep the logical resources abstract by hiding the definitions. In the implication proofs, we must construct separation logic resources used in the implied specification, given the abstract separation logic resources of the assumed specification, in

[1]The Coq proof assistant has recently been renamed to Rocq.

a way that lets us show the formal specifications for each of the individual operations of the implied specification. That is while hiding the fact that the newly constructed logical resources are defined in terms of the assumed logical resources. Lastly, to verify that the multi-version concurrency control algorithm of Berenson et al. [1995] implements our separation logic specifications, in the context of a distributed system, requires non-trivial and substantial proof effort (more than 5000 lines of Rocq proof code), especially since our specification for snapshot isolation includes evidence for unsuccessful commits in the form of a conflict between concurrent transactions expressed as a resource in the postcondition of the commit specification.

Contributions. In summary, our paper makes the following contributions:

(1) The first modular separation logic specifications for weak isolation levels, in particular read uncommitted, read committed and snapshot isolation. The specifications are independent of the underlying algorithm and whether the system is a local or distributed system, and we provide examples that show the specifications prohibit the phenomena of the literature.
(2) Formal proofs showing that the specification of snapshot isolation implies the specification of read committed, and that the specification of read committed implies the specification of read uncommitted.
(3) Implementation and verification of an in-memory single-node key-value database with snapshot isolation. The implementation is based on the original multi-version concurrency control snapshot isolation algorithm, and is the first formally verified executable database supporting transactions.

This gives us, for the first time, one unified logic in which we can verify clients and database implementations for transactions with weak isolation levels. That is all while enjoying the modularity of higher-order separation logic such as the ability to combine code using transactions with other verified libraries (something which existing work do not address). All results in this paper are mechanized in the Rocq proof assistant using the Iris separation logic.

Structure of the paper. The rest of this paper is structured as follows: In Section 2, we present our specification for read uncommitted using examples we have verified. Section 2.2 considers the proof of a particular example in detail using the read uncommitted specification. In Section 3, we present our specification for read committed and the examples we have proven using the read committed specification. In Section 4, we prove the specification for read committed implies the specification for read uncommitted. In Section 5, we present our specification for snapshot isolation. Here, we also present the examples we have proven using the snapshot isolation specification. In Section 6, we prove the specification for snapshot isolation implies the specification for read committed. In Section 7, we explain the verification of our database implementation with respect to our specification for snapshot isolation. In Section 8 we discuss related work, and in Section 9 we talk about future work before concluding. We will refer to Appendix A for presentation of a utility wait operation for all the isolation levels and a utility run operation for snapshot isolation together with a discussion of why a run operation is impractical for read uncommitted and read committed.

2 Specifying Read Uncommitted

In this section, we first recall the informal description of read uncommitted, and then present our modular separation logic specification in Section 2.1. This is done using the so-called *read uncommitted data* and *read own data* examples which we have verified using our specifications. Afterwards, in Section 2.2, we show in some detail how we can use the specification to prove safety and correctness of the concurrently executing transactions in the *read uncommitted data* example.

We present the proof to help the reader get acquainted with using the style of specifications presented throughout the paper.

Read uncommitted. Read uncommitted is the weakest isolation level. The level has mostly theoretical value, and it is often used in transactional models to establish a baseline, of definitions and structure, which we can use to define stronger levels. We follow the same approach. The only guarantees one gets from using read uncommitted is that data being read has been written by some transaction (captured by the *read uncommitted data* example in Figure 1) , and if a read comes after a write from the same transaction, then the read will be reading from this write (captured by the *read own data* example in Figure 2).

The *read uncommitted data* example (we ignore the invariant in blue for now) consists of two transactions (separated by the vertical bars), which are executed concurrently on two different nodes in a distributed system, alternatively threads in a local system, in which there is also a server with the key-value store. Notice that the example includes an assert statement, which crashes if the expression within it does not hold. We use assert because it makes it easy to state (in a logic-independent way) that an intended property holds. In this example, the assert statement expresses that the reading transaction will read nothing or the value written by the writing transaction, even though the latter does not commit. This is possible in read uncommitted as there is no guarantee that transactions will read committed data; it is only guaranteed that the data has been written by another transaction — independently of whether this transaction aborted or committed.

The *read own data* example follows a similar structure with two transactions. Here the assert statement expressed that no matter when the write of the first transactions is scheduled, the second transactions will always read the value previously written by itself.

$$
\begin{array}{l||l}
\mathsf{start} & \mathsf{start} \\
\mathsf{write}\ x\ 1 & v_x = \mathsf{read}\ x \\
\mathsf{loop} & \mathsf{assert}(v_x = \mathsf{None} \lor v_x = \mathsf{Some}\ 1) \\
 & \mathsf{commit}
\end{array}
$$

$$
Inv \triangleq \exists V,\ \textsc{ru}.x \mapsto V \ * (V = \emptyset \lor V = \{1\})
$$

Fig. 1. Read uncommitted data.

$$
\begin{array}{l||l}
 & \mathsf{start} \\
\mathsf{start} & \mathsf{write}\ x\ 2 \\
\mathsf{write}\ x\ 1 & v_x = \mathsf{read}\ x \\
\mathsf{commit} & \mathsf{assert}(v_x = \mathsf{Some}\ 2) \\
 & \mathsf{commit}
\end{array}
$$

Fig. 2. Read own data.

Some formalizations of read uncommitted, e.g., Adya [1999], imposes restrictions on the order in which versions are installed inside the key-value store. As separation logic abstracts away from implementation details, this is not a requirement we can or want to impose. This choice is in line with the work of Crooks et al. [2017] which takes a client centric approach.

2.1 Separation Logic Specifications

Our formal specifications of the read uncommitted operations are defined using the Aneris program logic [Krogh-Jespersen et al. 2020]. Aneris is a higher-order distributed separation logic, based on the Iris framework [Jung et al. 2018], fully mechanized in the Rocq proof assistant.[2] Our specifications are written using Hoare triples $\{P\}\ e\ \{v.\,Q\}^{ip}$. A Hoare triple asserts that given a machine state satisfying the precondition P, then the execution of the expression e is safe and, moreover, if the execution terminates then Q holds for the resulting machine state when the return value is bound to v. In Aneris, Hoare triples are annotated with the IP-address of the node on which the expression is executing, but we often omit this annotation when the IP-address is not of importance or for

[2]While we happened to use Aneris, we could have used another distributed separation logic based on the Iris framework such as Grove [Sharma et al. 2023], see the discussion in Section 8.

RU-INIT-CLIENT-SPEC
$$\{\mathsf{True}\}\ \mathsf{init\_client}\ srv\ \left\{c.\ \textsc{ru}.\mathsf{ConnectionState}(c, \mathsf{CanStart})\right\}$$

RU-INIT-KVS-SPEC
$$\left\{\textsc{ru}.\mathsf{InitKVS}\right\}\ \mathsf{init\_srv}\ \left\{().\ \mathsf{True}\right\}^{srv}$$

RU-READ-SPEC
$$\left\langle \textsc{ru}.k \mapsto_c ov * \textsc{ru}.k \mapsto V \right\rangle$$
$$\mathsf{read}\ c\ k$$
$$\left\langle \begin{array}{l} ow.\ \textsc{ru}.k \mapsto_c ov * \textsc{ru}.k \mapsto V * \\ \Big(\big(ov = \mathsf{None} \land \\ \quad ((\exists v, ow = \mathsf{Some}\ v \land v \in V) \lor \\ \quad\ ow = \mathsf{None})\big) \lor \\ \big(ov \neq \mathsf{None} \land ow = ov\big)\Big) \end{array} \right\rangle$$

RU-START-SPEC
$$\left\langle \textsc{ru}.\mathsf{ConnectionState}(c, \mathsf{CanStart}) * \mathop{\Large\ast}_{(k,V) \in m} \textsc{ru}.k \mapsto V \right\rangle$$
$$\mathsf{start}\ c$$
$$\left\langle \begin{array}{l} ().\ \textsc{ru}.\mathsf{ConnectionState}(c, \mathsf{Active}(\mathrm{dom}(m))) * \\ \mathop{\Large\ast}_{(k,V) \in m} \left(\textsc{ru}.k \mapsto V\ * \textsc{ru}.k \mapsto_c \mathsf{None}\right) \end{array} \right\rangle$$

RU-COMMIT-SPEC
$$\left\langle \begin{array}{l} \mathsf{ConnectionState}(c, \mathsf{Active}(S)) * S = \mathrm{dom}\ m = \mathrm{dom}\ mc * \\ \mathop{\Large\ast}_{(k,V) \in m} \textsc{ru}.k \mapsto V\ * \mathop{\Large\ast}_{(k,ov) \in mc} \textsc{ru}.k \mapsto_c ov \end{array} \right\rangle$$
$$\mathsf{commit}\ c$$
$$\left\langle v.\ \textsc{ru}.\mathsf{ConnectionState}(c, \mathsf{CanStart}) * \mathop{\Large\ast}_{(k,V) \in m} \textsc{ru}.k \mapsto V \right\rangle$$

RU-WRITE-SPEC
$$\left\langle \textsc{ru}.k \mapsto_c ov * \textsc{ru}.k \mapsto V \right\rangle$$
$$\mathsf{write}\ c\ k\ v$$
$$\left\langle \begin{array}{l} ().\ \textsc{ru}.k \mapsto_c \mathsf{Some}\ v * \\ \textsc{ru}.k \mapsto (V \cup \{v\}) \end{array} \right\rangle$$

Fig. 3. Specification for read uncommitted (RUSpecs).

presentation purposes. In fact, for most parts of this paper (except for the proof of the snapshot isolation implementation in Section 7), one can just pretend that the logic is the standard Iris logic; no special knowledge of Aneris is required to understand the paper. The propositions in the precondition and postcondition of the Hoare triples are often referred to as resources, since Aneris is a separation logic with a concept of ownership. We recall that the separating conjunction $P * Q$ only can be proven from a resource that can be split into two resources satisfying P and Q respectively. The magic wand $P \twoheadrightarrow Q$ is closely related to the separating conjunction: in combination with exclusive ownership of P, then $P \twoheadrightarrow Q$ entails Q. Some resources denote pure knowledge and do not assert exclusive ownership; this holds in particular for all persistent predicates, which are those satisfying that $\square P \vdash P$, where $\square$ is the so-called persistence modality which, intuitively, takes a predicate and forgets about the resources exclusively owned by the predicate. Persistent predicates can thus always be freely duplicated. For instance, Hoare triples are defined using the persistent modality, to allow us to use our specifications an unbounded number of times.

We now begin to describe the specifications of the operations of read uncommitted, which can all be found in Figure 3. We motivate and introduce the concepts used in the specifications along the way. For notation purposes, as we will be presenting specifications for other isolation levels with similar notation, we use the color blue and the prefix $\textsc{ru}.$ when presenting the read uncommitted specification. Later, we will use similar notation for the other isolation levels.

Init client operation. The specifications for all our isolation levels are centered around exclusive points-to resources. This is a popular choice in separation logic as it allows specifications only to require knowledge of a single key/location as opposed to a larger state. We will be working with two types of exclusive points-to resources: A *local points-to resource* per active transaction, denoted $\textsc{ru}.k \mapsto_c ov$ for read uncommitted, which asserts that the key k points to an optional value ov for the transaction active on connection c, and a *global points-to resource*, denoted $\textsc{ru}.k \mapsto V$ which asserts that the key k points to a state consisting of a set of values V, which captures the (global) state of the key-value store. A client only gets to make requests to the key-value store once a connection has been established with the provider of the key-value store, whether it is a local connection or happens over the network. A connection can be established by the client calling the init_client method, whose specification RU-INIT-CLIENT-SPEC is in Figure 3. A client can always try to establish a connection with the key-value store (the precondition is simply True), and in the postcondition, the client gains proof that it has a connection in the form of the resource $\mathsf{ConnectionState}(c, \mathsf{CanStart})$. The init_client method keeps trying to establish a connection to the server with IP-address *srv* when called, thus if no key-value store is available on the IP-address, the call will not terminate. If a client has multiple connections to different key-value stores, the points to resources will also take as argument the IP-address of the key-value store to which they belong. As long as there is only one active key-value store, the IP-address argument is implicit.

Start operation. As potentially many clients concurrently want to start their transactions, the resources capturing the key-value store state will often reside in an invariant. Invariants are propositions that hold at every step of execution once they have been established. An invariant holding a resource P is denoted $\boxed{P}$. [3] In Iris, and hereby Aneris, one can only open invariants across atomic expressions (expressions which in the operational semantics can step to a value using a single step). For this reason, we use *logical atomic triples* instead of standard Hoare triples to specify the transaction operations (for notation, logical atomic triples uses angle brackets as opposed to the curly brackets of ordinary Hoare triples). We refer to Jung et al. [2015] for a detailed description of logical atomic triples; in this paper, we simply treat them as special Hoare triples that allow us to open invariants around them. In Figure 4, we display the rule used to open an invariant around atomic Hoare triples.

$$\textsc{Inv-atomic}\quad \frac{\vdash \langle R * P\rangle\; e\; \langle v.\, Q * R\rangle}{\boxed{R} \vdash \langle P\rangle\; e\; \langle v.\, Q\rangle}$$

Fig. 4. Invariant rule for atomic Hoare Triples.

With the rule INV-ATOMIC, we can use the resources in an invariant to prove the precondition of an atomic Hoare Triple, and we can use the resources in the postcondition of the atomic Hoare Triple to reestablish the invariant.

Now, the RU-START-SPEC rule takes as precondition the connection state together with global points-to resources for all the keys on which the transaction wants to perform reads and writes – these are the keys in the map m in the notation. As the key-value store state, in the form of global points-to resources, can be shared among many clients in an invariant, the start operation is specified using a logical atomic triple. The postcondition of the start operation states that the connection is active for the keys in the domain of m (this information will be used later at commit time) and gives back the global points-to resources together with fresh local points-to resources.[4]

Now, we could have elided an explicit init_client operation and made it part of the start operation to simplify the specification. This would, however, imply initialization overhead (cache and network-connection creation) on a per-transaction basis which is suboptimal.

[3]A reader familiar with Iris will notice that we have omitted invariant names, and in later parts of the paper masks, to make the presentation less technical.

[4]Separation-logic aficionados who are familiar with the frame rule may wonder why the global points-to resources are included unchanged in both the precondition and the postcondition; we have decided to include them (both for start and commit) in order to be able to prove formally that read committed implies read uncommitted, see Section 4.

Read and write operation. Having started a transaction with the start operation, an active transaction can perform read and write operations using its newly gained local points-to resources. The read operation requires the global and local points-to resources for the key in its precondition. If the active transaction has already written to the key, this value will be read straight from the local points-to resource. Otherwise, the value being read can be None or come from the global state, i.e., the value will be from the set of values V pointed to by the global points-to resource. As for the read operation, the precondition of the specification for the write operation requires global and local points-to resources for the key that is being written to. As we are specifying read uncommitted where transactions are allowed to read uncommitted data, both the global and local points-to resource is updated. Note, in particular, that the global points-to resource gets updated immediately for other transactions to read, even if the writing transaction does not commit.

At this point, it can be tempting to create a stronger specification. For instance, a meaningful optimization would be to cache a local read and update ov to ow when ov is none for local points-to resource in the read specification. But if we enforce such an optimization in the specification, then we are specifying something stronger than read uncommitted (not all implementations of read uncommitted have this optimization). Do notice that any implementation doing the optimization can implement our specification, it will simply pick ow from the set V on a subsequent read if ov remains None.

Commit operation. At commit time, the connection state, local and global points to resources are collected in the precondition of the RU-COMMIT-SPEC rule. The domain conditions ensure that local and global points-to resources are available for all the keys on which the transaction was started. No matter if the commit is successful or not (the return value of commit is a boolean), the state of the key-value store remains the same in the postcondition of the RU-COMMIT-SPEC, as all the updates made by the transaction have already been propagated at the time of the write operation, cf. RU-WRITE-SPEC, to the global points-to resources. Thus, the job of the commit specification in read uncommitted is simply to collect the local points-to resources and change the connection state.

Init KVS operation and the complete specification. Initializing the key-value store with RU-INIT-KVS-SPEC requires the $_{\text{RU}}.\mathsf{InitKVS}$ resources which can not be obtained from using the per operation specifications in Figure 3. Likewise, initial global points to resources must be obtained before one can start using the specifications in Figure 3. Thus, a complete specification for an isolation level must provide all the resources we need in order to get started with using the per operation specifications. Below, in (1), we have defined exactly what the complete read uncommitted specification RUSpec is. The complete specification RUSpec is stated using the Iris update modality. The Iris update modality $\Rrightarrow P$ states that initialization of resources or updates to currently owned resources can be made to satisfy P. In the context of RUSpec, this means we can initialize the resources that constitute the specification. We will continue with explaining each of the components RUSpec consists of.

$$\mathsf{RUSpec} \triangleq \Rrightarrow {}_{\text{RU}}.\mathsf{InitKVS} * \mathop{\Huge *}_{k \in \mathsf{Keys}} {}_{\text{RU}}.k \mapsto \emptyset * \mathsf{RUSpecs} * {}_{\text{RU}}.\boxed{\mathsf{GlobalInv}} \tag{1}$$

First, we have the resource $_{\text{RU}}.\mathsf{InitKVS}$ to initialize the key-value store as mentioned. Next, for every key we have $_{\text{RU}}.k \mapsto \emptyset$ with each key pointing to the empty set as the initial state. RUSpecs is defined as all the specifications from Figure 3, concretely it is the separated conjunction between each of the Hoare Triples in Figure 3. Lastly, we have a global invariant $_{\text{RU}}.\boxed{\mathsf{GlobalInv}}$, which we have not yet seen a use case for. The global invariant is needed to be able to prove two properties which we have named the *exclusion* and *creation* properties. We will first motivate and then explain these properties next.

The exclusion and creation properties. As already mentioned, the state of key-value store, in the form of global points-to resources, will be shared using an invariant when concurrent access is needed (see the following subsection for an example). The invariant has to be of a form that captures the key-value store states that are created by the transactions. Not all states will be valid at any given time, for instance, if a value has been observed it can not disappear later. To capture such observations, our specifications (for all our isolation levels) come with a *seen* resource, which works together with the global points-to resource. The $\mathsf{Seen}(k, V)$ resource for read uncommitted intuitively captures the observation of having seen the key-value store in the state where k was pointing to the set of values V. Clients can use this knowledge to exclude absurd states of an invariant. The exclusion of key-value store states based on observations happens through the exclusion property (2).

$$\textsc{ru}.\boxed{\mathsf{GlobalInv}} * \textsc{ru}.\mathsf{Seen}(k, V') * \textsc{ru}.k \mapsto V \vdash \Rrightarrow \textsc{ru}.k \mapsto V * V' \subseteq V \tag{2}$$

$$\textsc{ru}.\boxed{\mathsf{GlobalInv}} * \textsc{ru}.k \mapsto V \vdash \Rrightarrow \textsc{ru}.k \mapsto V * \textsc{ru}.\mathsf{Seen}(k, V) \tag{3}$$

The essence of the exclusion property is that given an observation $\textsc{ru}.\mathsf{Seen}(k, V')$ and the current state of a key $\textsc{ru}.k \mapsto V$, it must be the case that the current state is the same as the observation or the result of having applied updates to the observation, i.e., $V' \subseteq V$ holds. The seen resources are created by the creation property (3). To be able to use the creation and the exclusion properties we must have access to the global invariant of the read uncommitted key-value library which is therefore a part of the complete specification RUSpec. [5]

We remark that in the exclusion property (2) and the creation property (3), the $\textsc{ru}.\boxed{\mathsf{GlobalInv}}$ and $\textsc{ru}.\mathsf{Seen}(k, V')$ resources are not included on the right-hand sides but can be found on the left-hand sides of the entailment; it is not necessary to include them on the right since both of these predicates are persistent and hence can be freely duplicated prior to applying the exclusion property rule.

2.2 Proof of Read Uncommitted Data Example

In this section, we show how to use the specifications from Figure 3 to prove the *read uncommitted data* example shown in Figure 1. What we mean by "proving the read uncommitted data example" is that we prove a specification (a Hoare triple) for this example. By adequacy of the Aneris logic, this then means that the assert shown in Figure 1 will not fail. In other words, by proving a specification for the example, we show that the property expressed in the assert statement holds for any interleaving.

We begin by using the setup rule (1). As the key-space for the example consists only of x, the setup rule gives us $\textsc{ru}.x \mapsto [\,]$. This resource is then used to create the client invariant $\boxed{\exists V,\ x \mapsto V * (V = \emptyset \lor V = \{1\})}$, also shown in Figure 1 (all the examples in this paper uses invariants but we only state them when it is relevant for the presentation). The $\textsc{ru}.\mathsf{InitKVS}$ resource from the setup is used to initialize the key-value store using ru-init-kvs-spec. We then prove each of the two client transactions separately — this is possible in Aneris as long as one distributes resources appropriately among the clients prior to doing the individual proofs. As invariants are persistent and thus duplicable resources, we can create two copies of $\boxed{\exists V,\ x \mapsto V * (V = \emptyset \lor V = \{1\})}$ and use one for each of the individual clients proofs. The sharing of invariants is only safe as long as

[5]The Seen resources can also be defined as a standalone library that will assume the remaining read uncommitted specification.

each client owning an invariant uses it safely (this is enforced in Iris). Namely, the resources in an invariant can be accessed around operations specified with atomic Hoare triples or atmoic expressions where we use the former. For both transactions it is the case that their proofs are bootstrapped using RU-INIT-CLIENT-SPEC, which means that in each case, we have a $_{\text{RU}}.\mathsf{ConnectionState}(c, \mathsf{CanStart})$ resource for their connection c (each client have a unique connection). We will now proceed by proving the clients transactions one at a time.

Write-transaction. The transaction is started using the start operation. To satisfy the precondition of the RU-START-SPEC rule, we provide $_{\text{RU}}.\mathsf{ConnectionState}(c, \mathsf{CanStart})$ and the global points-to resources from the client invariant, i.e., $_{\text{RU}}.x \mapsto V$ for a fresh set of values V for which $V = \emptyset \vee V = \{1\}$ holds. By the RU-START-SPEC rule we then obtain the resources $_{\text{RU}}.\mathsf{ConnectionState}(c, \mathsf{Active}(\{x\})) * {}_{\text{RU}}.x \mapsto V * {}_{\text{RU}}.x \mapsto_c \mathsf{None}$. The global points-to resource $_{\text{RU}}.x \mapsto V$ is used to close the client invariant while we hold onto the rest of the resources. We then use the RU-WRITE-SPEC rule for the write x 1 operation. Again, we must open the invariant to gain $_{\text{RU}}.x \mapsto V'$, for a fresh set of values V' for which $V' = \emptyset \vee V' = \{1\}$ holds. Now, $_{\text{RU}}.x \mapsto V'$, together with the local points-to resource $_{\text{RU}}.x \mapsto_c \mathsf{None}$ we obtained using the RU-START-SPEC rule, is provided to the RU-WRITE-SPEC rule to obtain $_{\text{RU}}.x \mapsto V' \cup \{1\} * {}_{\text{RU}}.x \mapsto_c \mathsf{Some}\ 1$. Notice that no matter which of the cases is true in the disjunction $V' = \emptyset \vee V' = \{1\}$, we have that $V' \cup \{1\}$ equals $\{1\}$ which is sufficient for restoring the client invariant again. We can hold on to $_{\text{RU}}.x \mapsto_c \mathsf{None}$, but we will no longer need it at this point. The last operation is loop, which is a non-terminating operation and which therefore trivially satisfies any specification (as we are using a partial-correctness logic; technically, in Iris this is proved using so-called Löb induction).

Read-transaction. As for the write-transaction, we initialize using the RU-INIT-CLIENT-SPEC rule, and thus we obtain ownership of the resource $_{\text{RU}}.\mathsf{ConnectionState}(c, \mathsf{CanStart})$. Also, as for the write-transaction, reasoning about the start operation goes the same way by opening and closing the client invariant leaving us to continue with the resources $_{\text{RU}}.\mathsf{ConnectionState}(c, \mathsf{Active}(\{x\})) * {}_{\text{RU}}.x \mapsto_c \mathsf{None}$. For the read operation read x , we must open the client invariant to gain $_{\text{RU}}.x \mapsto V$ for a fresh set of values V for which $V = \emptyset \vee V = \{1\}$ holds. According to the postcondition of RU-READ-SPEC, as our local points-to resource points to None, we get the following information about the return value ow of the write operation: $(\exists v, ow = \mathsf{Some}\ v \wedge v \in V) \vee ow = \mathsf{None}$. Having obtained this information we can close the invariant with $_{\text{RU}}.x \mapsto V$ and proceed to the assertion assert$(v_x = \text{None} \vee v_x = \text{Some } 1)$. As v_x is equal to the return value of the write operation ow, we use the information $(\exists v, ow = \mathsf{Some}\ v \wedge v \in V) \vee ow = \mathsf{None}$ together with our knowledge about the set V, i.e., $V = \emptyset \vee V = \{1\}$, to conclude that the assertion does indeed hold. Having concluded that the assertion holds, we are only left with reasoning about commit. In the precondition of RU-COMMIT-SPEC we collect the connection state, $_{\text{RU}}.\mathsf{ConnectionState}(c, \mathsf{Active}(\{x\}))$, the local points-to resource, $_{\text{RU}}.x \mapsto_c \mathsf{None}$, and the global points-to resource from the invariant for a fresh V', $_{\text{RU}}.x \mapsto V'$. The postcondition gives us back $_{\text{RU}}.x \mapsto V'$ to close the invariant and the ability to start a new transaction in the form of the connection state $_{\text{RU}}.\mathsf{ConnectionState}(c, \mathsf{CanStart})$.

Remarks. Notice how the proof of the read uncommitted data example is modular: We use the node-local reasoning of Aneris, together with the client invariant, to reason about each transaction in isolation. The proof is also modular in the sense that it only relies on the specification of read uncommitted, not the implementation thereof. We further remark that it follows formally by the implication proofs in Sections 4 and 6 that the example is also provable using read committed and snapshot isolation.

3 Specifying Read Committed

In this section, we present our modular separation logic specification for read committed. We also present the *dirty read* example and the *commit order* example which we have proven using the specification. Contrary to read uncommitted, read committed is of high practical value as it is the default isolation level of leading database systems such as Microsoft SQL, Oracle or PostgreSQL. A reader unfamiliar with transactional guarantees may be surprised by the wide spread of read committed as, albeit being stronger than read uncommitted, read committed does not impose any order on the data being read. Instead, read committed builds upon read uncommitted by adding the requirement that transactions are only allowed to read committed data. Many formalizations specifying read committed, and other isolation levels for that sake, are not concerned with guarantees about aborting transaction. For example, Adya [1999]; Crooks et al. [2017] do not require that uncommitted transactions must read committed data. Allowing aborting transactions not to read committed data would significantly increase the complexity of our separation logic specification, and hence we have made the design choice that aborting transactions should also read committed data. In practice, imposing the requirement that uncommitted transactions may read *any* uncommitted data is a weak requirement that will not exclude many, if any, implementations from realizing the specification. We remark that our specification still specifies read committed, it is just not the weakest specification there exists (an algorithm producing a subset of the valid executions under an isolation level does naturally still adhere to that isolation level). We remark that an equivalent choice is made in Chang et al. [2023] when specifying serializability.

RC-INIT-CLIENT-SPEC

$$\{\mathsf{True}\}\ \mathsf{init\_client}\ srv\ \{c.\ \textsc{rc}.\mathsf{ConnectionState}(c, \mathsf{CanStart})\}$$

RC-INIT-KVS-SPEC

$$\{\textsc{rc}.\mathsf{InitKVS}\}\ \mathsf{init\_srv}\ \{().\ \mathsf{True}\}^{srv}$$

RC-READ-SPEC

$$\langle \textsc{rc}.k \mapsto_c ov * \textsc{rc}.k \mapsto V \rangle$$
$$\mathsf{read}\ c\ k$$
$$\left\langle \begin{array}{l} ow.\ \textsc{rc}.k \mapsto_c ov * \textsc{rc}.k \mapsto V * \\ \Big(\big(ov = \mathsf{None} \land \\ \quad ((\exists v, ow = \mathsf{Some}\ v \land v \in V) \lor \\ \quad\quad ow = \mathsf{None})\big) \lor \\ \big(ov \neq \mathsf{None} \land ow = ov\big)\Big) \end{array} \right\rangle$$

RC-START-SPEC

$$\left\langle \textsc{rc}.\mathsf{ConnectionState}(c, \mathsf{CanStart}) * \mathop{\Huge *}_{(k,V)\in m} \textsc{rc}.k \mapsto V \right\rangle$$
$$\mathsf{start}\ c$$
$$\left\langle \begin{array}{l} ().\ \textsc{rc}.\mathsf{ConnectionState}(c, \mathsf{Active}(\mathsf{dom}(m))) * \\ \mathop{\Huge *}_{(k,V)\in m} \big(\textsc{rc}.k \mapsto V * \textsc{rc}.k \mapsto_c \mathsf{None}\big) \end{array} \right\rangle$$

RC-COMMIT-SPEC

$$\left\langle \begin{array}{l} \textsc{rc}.\mathsf{ConnectionState}(c, \mathsf{Active}(S)) * S = \mathsf{dom}\ m = \mathsf{dom}\ mc * \\ \mathop{\Huge *}_{(k,V)\in m} \textsc{rc}.k \mapsto V \ * \mathop{\Huge *}_{(k,ov)\in mc} \textsc{rc}.k \mapsto_c ov \end{array} \right\rangle$$
$$\mathsf{commit}\ c$$
$$\left\langle \begin{array}{l} v.\ \mathsf{ConnectionState}(c, \mathsf{CanStart}) * \\ \Big(v = \mathsf{True} * \mathop{\Huge *}_{\substack{(k,V)\in m \\ (k,ov)\in mc}} \textsc{rc}.k \mapsto update\_vals(ov, V)\Big) \lor \\ \Big(v = \mathsf{False} * \mathop{\Huge *}_{(k,V)\in m} \textsc{rc}.k \mapsto V\Big) \end{array} \right\rangle$$

RC-WRITE-SPEC

$$\langle \textsc{rc}.k \mapsto_c ov \rangle$$
$$\mathsf{write}\ c\ k\ v$$
$$\langle ().\ \textsc{rc}.k \mapsto_c \mathsf{Some}\ v \rangle$$

Fig. 5. Specification for read committed (RCSpecs).

Specification. The specification (Figure 5) is based upon the same structure as the specification for read committed: local and global points-to resources for sets of values with the only differences occurring in the RC-WRITE-SPEC and the RC-COMMIT-SPEC rules. Hence, we present the read committed specification by highlighting the difference from the read uncommitted specification in the RC-WRITE-SPEC and the RC-COMMIT-SPEC rules.

In comparison with RU-WRITE-SPEC, RC-WRITE-SPEC does not update the global points-to resource, and the update is simply recorded in the local points-to resource. This captures that a concurrently active transaction can not read the change until it is committed. In the RC-COMMIT-SPEC, if the commit is successful, the updates recorded in the local points-to resources are added to the global points-to resources using the *update_vals* function, see (4). In case the commit is unsuccessful, the key-value store state, i.e., the global points-to resources, remain unchanged and the changes of the transaction are lost.

$$update\_vals(ov, V) \triangleq \text{match } ov \text{ with Some } v \Rightarrow V \cup v \mid \text{None} \Rightarrow V \tag{4}$$

As for read uncommitted, observations can be made using Seen resources. In fact, the exclusion property (2) and the creation property (3) holds for read committed too (all resources should be prefixed with RC. instead of RU.). To conclude the presentation of the read committed specification, we state the complete specification, similar to (1), which formally defines the read committed specification (RCSpec), see (5).

$$\text{RCSpec} \triangleq \Rrightarrow {}_{\text{RC}}.\text{InitKVS} * \mathop{\Large *}_{k \in \text{Keys}} {}_{\text{RC}}.k \mapsto \emptyset * \text{RCSpecs} * {}_{\text{RC}}.\boxed{\text{GlobalInv}} \tag{5}$$

Read Committed Example. We have used the read committed specification to formally prove the *dirty read* example Figure 6 and the *commit order* example Figure 7.

start	start
write x 1	v_x = read x
loop	assert(v_x = None)
	commit

$$Inv \triangleq \exists V,\ {}_{\text{RC}}.x \mapsto V \ * V = \emptyset$$

Fig. 6. Dirty read example.

The dirty read example builds upon the read uncommitted data example from Figure 1, and is based upon the dirty read phenomenon in the literature (Adya [1999]; Berenson et al. [1995] and 1992 ANSI SQL standard), by removing the clause in the conjunction of the assertion that corresponds to the reading transaction seeing the uncommitted data from the writing transaction. This is as expected for read committed, since read committed disallows transactions to read uncommitted data. Recall that a phenomenon is an example of an execution between concurrently executing transactions that should be prohibited; in our example this is captured by the assertion showing that it is impossible. The proof of the example is simple and uses the invariant shown in Figure 6 which state that no value will be written to the x variable.

The commit order example asserts the absence of a situation in which two transactions both are seeing data committed by the other transaction. This should not be allowed to occur as there must be an order in which transactions commit. The first two transactions in the dirty read example revolve around the same structure: First the value 1 is written into a key that the other transaction reads from and checks whether it was written to using an if-statement. If this was the case, the transactions write to another key (respectively a and b) to signify that they saw the write of the other transaction. The last transaction reads from a and b and asserts that the transactions could not possible both have seen each others writes (the case where both a and b has been written to).

start	start	start
write x 1	write y 1	v_a = read a
v_y = read y	v_x = read x	v_b = read b
If (v_y = Some 1) then	If (v_x = Some 1) then	assert(!(v_a = Some 1 $\wedge$ v_b = Some 1))
{write a 1 }	{write b 1 }	commit
commit	commit	

Fig. 7. Commit order example.

We remark that it follows from the implication proof in Section 6 that the two examples in this section are also provable for snapshot isolation.

4 Read Committed Implies Read Uncommitted

Given our formal separation logic specifications of read uncommitted and read committed, we can now formally prove that read committed implies read uncommitted:

Theorem 4.1 (Read committed implies read uncommitted). *The specification for read committed* (5) *implies the specification for read uncommitted* (1)*:*

$$\mathsf{RCSpec} \longrightarrow \mathsf{RUSpec}.$$

Proof Structure. The proof, consisting of approximately 800 lines of Rocq proof code, centers around closing the gap between read uncommitted and read committed that comes from the fact that in read uncommitted the global state is updated immediately at the time of writing values, whereas for read committed, the global state is not updated until the transaction commits successfully. The proof proceeds naturally by first assuming the read committed specification RCSpec (5), which consists of the global invariant $\textsc{rc}.\boxed{\mathsf{GlobalInv}}$, fresh global points-to resources $\ast_{(k,V)\in\mathsf{Keys}}\ \textsc{rc}.k \mapsto \emptyset$, an initialization resource $\textsc{rc}.\mathsf{InitKVS}$ for the key-value store and the read committed per operation specifications RCSpecs (Figure 5). To prove RUSpec, we must naturally show the equivalent resources and specifications for read uncommitted (including the exclusion (2) and creation (3) properties). Now, we need to define all the resources used in RUSpec. In doing this, we will have all the resources found in RCSpec at our disposal. We define the key-value store resource for initialization to be the same as the corresponding read committed resource, but for the global invariant and the global and local points-to resources we use a more involved definition with *resource algebras*. Resource algebras are the building blocks of Iris, see Jung et al. [2016] for details. All we need to know here is that using resource algebras, we can create custom predicates and rules to relate these predicates (usually we will refer to these custom predicates as resources algebras). Having defined all resources, the largest part of the proof revolves around showing the per operation specifications of read uncommitted, i.e., RUSpecs. When proving each specification, we get to use the corresponding specification for read committed from RCSpecs as an assumption. This is helpful, because we first assume the read uncommitted resources in the precondition of the specification and these we have defined in terms of the read committed resources and resource algebras. We thus use the read committed specification for the particular operation to transform the read committed resources, while the resource algebras are transformed by hand, to meet the post condition of the read uncommitted specification.

We remark that while the proof follows a quite natural path, it is non-trivial to come up with a definition of the read uncommitted resources in terms of the read committed resources that makes the proof work. We refer to the Rocq formalization for the details of the whole proof and continue here with an excerpt regarding the write specification and insights about how the proof goes at commit time for the commit specification.

Proof Excerpt. Our goal is to prove the RU-WRITE-SPEC rule which uses the resources $_{\text{RU}}.x \mapsto V$ and $_{\text{RU}}.k \mapsto_c ov$. We get to assume the RC-WRITE-SPEC rule and can define the read uncommitted resources, which we will refer to as the *wrapped* resources, using the equivalent resources for read committed, i.e., $_{\text{RC}}.x \mapsto V$ and $_{\text{RC}}.k \mapsto_c ov$.

To define the wrapped resources, we use a *ghost theory*, which is a set of separation logic predicates defined using resource algebras, together with rules that relate the different predicates. Our ghost theory consists of two predicates, $\mathsf{AuthSet}(V)$ and $\mathsf{FragSet}(V)$, on sets of values V. Intuitively, the *authorative* part $\mathsf{AuthSet}(V)$ can be initialized with the empty set, i.e., $V = \emptyset$, and will at all times hold all the values that have been added since V was the empty set. This is seen in (6) where, given the global invariant and $\mathsf{AuthSet}(V)$, one can make an update (signified by $\Rrightarrow$) by adding an element to V resulting in $\mathsf{AuthSet}(V \cup v)$. We remark that unless explicitly stated otherwise, resources from an invariant in Iris can be used to prove a proposition below the update modality as long as the invariant still holds after the update.

$$_{\text{RU}}.\boxed{\mathsf{GlobalInv}} * \mathsf{AuthSet}(V) \vdash \Rrightarrow {}_{\text{RU}}.\boxed{\mathsf{GlobalInv}} * \mathsf{AuthSet}(V \cup v) * \mathsf{FragSet}(V \cup v) \tag{6}$$

At the same time, one gains the *fragmental* part $\mathsf{FragSet}(V \cup v)$. As such, $\mathsf{FragSet}(V)$, for any V, represents the values of $\mathsf{AuthSet}(V)$ at some point in the past (note that (6) does not update old fragmental parts when new values are added, and values can not be removed). Naturally, this means that at any given point, we can conclude that the set of values in a fragmental resource is a subset of the values in an authoritative resource, see (7).

$$_{\text{RU}}.\boxed{\mathsf{GlobalInv}} * \mathsf{AuthSet}(V) * \mathsf{FragSet}(V') \vdash {}_{\text{RU}}.\boxed{\mathsf{GlobalInv}} * \mathsf{AuthSet}(V) * \mathsf{FragSet}(V') * V' \subseteq V \tag{7}$$

Having defined our ghost theory, we can define the wrapped resources used in the write specification Figure 8. Recall, $_{\text{RU}}.k \mapsto_c ov$ and $_{\text{RC}}.k \mapsto_c ov$ are used in the same way in read uncommitted and

$$_{\text{RU}}.x \mapsto V \triangleq \exists V', V' \subseteq V * {}_{\text{RC}}.x \mapsto V' * \mathsf{AuthSet}(V)$$
$$_{\text{RU}}.k \mapsto_c ov \triangleq \exists V, ((\exists v, ov = \mathsf{Some}\ v \wedge v \in V) \vee ov = \mathsf{None}) * {}_{\text{RC}}.k \mapsto_c ov * \mathsf{FragSet}(V)$$

Fig. 8. Wrapped resources.

read committed: ov/ov is the latest value written (if a write has been made) by the current active transaction, and it is updated using the write specification. In contrast, $_{\text{RU}}.x \mapsto V$ and $_{\text{RC}}.x \mapsto V$, holding the global state, are not used in the same way: For read uncommitted the global state is updated immediately at the time of writing the value, and for read committed the global state is not updated until the transaction commits successfully. Therefore, the wrapped resources for read uncommitted in Figure 8 are constructed to hide the fact that values are not propagated to the global state until commit time. To see this, observe that inside the definition of $_{\text{RU}}.x \mapsto V$ lies $_{\text{RC}}.x \mapsto V'$ with V' being a subset of V. The set V contains the committed and uncommitted values whereas V' only contains the committed values. $\mathsf{AuthSet}(V)$ is then used to keep track of all the uncommitted values. Further, $_{\text{RU}}.k \mapsto_c ov$ is defined using the equivalent resource $_{\text{RC}}.k \mapsto_c ov$ where it is also stated that if $ov = \mathsf{Some}\ v$ for some value v, then v must be in the set of all values (captured using $\mathsf{FragSet}(V)$ and the requirement $v \in V$ for some V). In the proof of the write specification, which we will sketch next, it is not evident that we the need our ghost theory since we only update it, but we do not use it to draw any conclusions. The necessity of the ghost theory appears at commit time: As V' in $_{\text{RC}}.x \mapsto V'$ is updated with v from $_{\text{RC}}.k \mapsto_c \mathsf{Some}\ v$ at commit time for read committed, and $_{\text{RU}}.x \mapsto V$ remains the same at commit time for read uncommitted, we use the fact that there exists some V'' with $v \in V''$, for which $\mathsf{FragSet}(V'')$ holds, per the definition of our wrapped resource, which implies v is also in V using rule (7) of our ghost theory (the resource $\mathsf{AuthSet}(V)$ comes from inside the definition of $\textsc{ru}.x \mapsto V$). Thus, at commit time, even though $\textsc{rc}.x \mapsto V'$ changes inside the wrapped resource $\textsc{ru}.x \mapsto V$, we can conclude that the added value is already in V, and V can remain unchanged, due to the ghost theory construction, which is needed to prove RU-COMMIT-SPEC. In this sense, the wrapped resource are constructed to hide the fact that values are not propagated until commit time.

The write specification we end up having to prove (when unwrapping the wrapped resources) is shown in Figure 9.

$$\left\langle \begin{array}{l}(\exists V, ((\exists v', ov = \mathsf{Some}\ v' \land v' \in V) \lor ov = \mathsf{None}) * \textsc{rc}.k \mapsto_c ov * \mathsf{FragSet}(V)) * \\ (\exists V', V' \subseteq V * \textsc{rc}.x \mapsto V' * \mathsf{AuthSet}(V))\end{array} \right\rangle$$
$$\mathsf{write}\ c\ k\ v$$
$$\left\langle \begin{array}{l}().\ (\exists V, ((\exists v', \mathsf{Some}\ v = \mathsf{Some}\ v' \land v' \in V) \lor \mathsf{Some}\ v = \mathsf{None}) * \textsc{rc}.k \mapsto_c \mathsf{Some}\ v * \mathsf{FragSet}(V)) * \\ (\exists V', V' \subseteq V \cup v * \textsc{rc}.x \mapsto V' * \mathsf{AuthSet}(V \cup v))\end{array} \right\rangle$$

Fig. 9. Unwrapped write specification.

In the precondition, when instantiating the existential quantifiers, the relevant resources we get to assume are $\textsc{rc}.k \mapsto_c ov * \textsc{rc}.x \mapsto V' * \mathsf{AuthSet}(V)$ with $V' \subseteq V$. Using the read committed write specification RC-WRITE-SPEC, we can transform the resource $\textsc{rc}.k \mapsto_c ov$ into $\textsc{rc}.k \mapsto_c \mathsf{Some}\ v$. Using (6), we can transform $\mathsf{AuthSet}(V)$ into $\mathsf{AuthSet}(V \cup v) * \mathsf{FragSet}(V \cup v)$. Hence, we can now satisfy the postcondition of Figure 9, using $\textsc{rc}.k \mapsto_c \mathsf{Some}\ v * \textsc{rc}.x \mapsto V' * \mathsf{AuthSet}(V \cup v) * \mathsf{FragSet}(V \cup v)$, which concludes the proof sketch of the write specification.

5 Specifying Snapshot Isolation

In this section, we first present our modular separation logic specification for snapshot isolation, and then we present three examples based on phenomena from the literature, *write skew*, *read skew* and *non-repeatable read*, in addition to a bank transfer example which we have all proven using the specification. Snapshot isolation, like read committed, is an isolation level with practical value, but it comes with much stronger guarantees about what transactions are allowed to read.

Snapshot isolation, as presented in Berenson et al. [1995], makes use of snapshots. A snapshot is defined as the state of the database at a given time. Snapshot isolation makes a distinction between the start snapshot, on which a transaction reads, and the commit snapshot, on which the transaction attempts to commit its updates. All commits happen in a globally defined linear order. The start and commit snapshots are taken at the time of the start operation and the commit operation respectively. In this paper, the snapshots will be the most recent snapshots at the time they are a captured. This can be referred to as "strong snapshot isolation" [Daudjee and Salem 2006], as one can relax the criteria on the start snapshot to be any valid snapshot at the time of the start operation — not necessarily the most recent. [6] Naturally, the start snapshot and the commit snapshot can be different as other concurrent transactions can commit in the time between the snapshots. This leads us to the crucial commit-check of snapshot isolation, which expresses when a transaction can commit:

[6]In Berenson et al. [1995] starting snapshots are allowed to be any previous snapshot and not necessarily the most recent. Specifications for the weaker variant would be much harder to work with, as you would get a disjunction of all possible snapshots to reason about when starting a transaction.

> *(Snapshot Isolation Commit-check)* A transaction is allowed to commit only if it has no write conflicts with any transaction committed between its own start snapshot and its commit snapshot.

As for the previous isolation levels, our separation logic specification will be based around local and global points-to resources, but this time the global points-to resources do not point to a set of values but *histories*. Histories are lists of values with the last element representing the latest committed value to the key. Histories, as opposed to sets, are needed to express the snapshot isolation commit check: we can use the histories from a start snapshot and a commit snapshot to conclude whether writes have happened between the two snapshots.

The initialization specifications, SI-INIT-CLIENT-SPEC and SI-INIT-KVS-SPEC, are identical to the ones of the previous isolation levels. We now describe the specifications of the start, read, write and commit operations for snapshot isolation which can all be found in Figure 10.

SI-INIT-CLIENT-SPEC
$$\{\mathsf{True}\}\ \mathsf{init\_client}\ srv\ \{c.\ {\scriptstyle \mathrm{SI}}.\mathsf{ConnectionState}(c, \mathsf{CanStart})\}$$

SI-INIT-KVS-SPEC
$$\{{\scriptstyle \mathrm{SI}}.\mathsf{InitKVS}\}\ \mathsf{init\_srv}\ \{().\ \mathsf{True}\}^{srv}$$

SI-START-SPEC
$$\Big\langle {\scriptstyle \mathrm{SI}}.\mathsf{ConnectionState}(c, \mathsf{CanStart}) * \mathop{\Large\ast}_{(k,h)\in m} {\scriptstyle \mathrm{SI}}.k \mapsto h \Big\rangle$$
$$\mathsf{start}\ c$$
$$\left\langle \begin{array}{l} ().\ {\scriptstyle \mathrm{SI}}.\mathsf{ConnectionState}(c, \mathsf{Active}(m)) * \\ \mathop{\Large\ast}_{(k,h)\in m} \Big({\scriptstyle \mathrm{SI}}.k \mapsto h * {\scriptstyle \mathrm{SI}}.k \mapsto_c last(h) * \\ \qquad\qquad {\scriptstyle \mathrm{SI}}.\mathsf{KeyUpdStatus}(c, k, \mathsf{False})\Big) \end{array} \right\rangle$$

SI-WRITE-SPEC
$$\left\langle \begin{array}{l} {\scriptstyle \mathrm{SI}}.k \mapsto_c ov * \\ {\scriptstyle \mathrm{SI}}.\mathsf{KeyUpdStatus}(c, k, b) \end{array} \right\rangle$$
$$\mathsf{write}\ c\ k\ v$$
$$\left\langle \begin{array}{l} ().\ {\scriptstyle \mathrm{SI}}.k \mapsto_c \mathsf{Some}\ v * \\ {\scriptstyle \mathrm{SI}}.\mathsf{KeyUpdStatus}(c, k, \mathsf{True}) \end{array} \right\rangle$$

SI-COMMIT-SPEC
$$\left\langle \begin{array}{l} {\scriptstyle \mathrm{SI}}.\mathsf{ConnectionState}(c, \mathsf{Active}(ms)) * \mathsf{dom}\ m = \mathsf{dom}\ ms = \mathsf{dom}\ mc * \\ \mathop{\Large\ast}_{(k,h)\in m} {\scriptstyle \mathrm{SI}}.k \mapsto h * \mathop{\Large\ast}_{(k,(ov,b))\in mc} \big({\scriptstyle \mathrm{SI}}.k \mapsto_c ov * {\scriptstyle \mathrm{SI}}.\mathsf{KeyUpdStatus}(c, k, b)\big) \end{array} \right\rangle$$
$$\mathsf{commit}\ c$$
$$\left\langle \begin{array}{l} v.\ {\scriptstyle \mathrm{SI}}.\mathsf{ConnectionState}(c, \mathsf{CanStart}) * \\ \Big(v = \mathsf{True} * {\scriptstyle \mathrm{SI}}.can\_commit(m, ms, mc) * \mathop{\Large\ast}_{\substack{(k,h)\in m \\ (k,p)\in mc}} {\scriptstyle \mathrm{SI}}.k \mapsto update\_hist(p, h)\Big) \vee \\ \Big(v = \mathsf{False} * \neg {\scriptstyle \mathrm{SI}}.can\_commit(m, ms, mc) * \mathop{\Large\ast}_{(k,h)\in m} {\scriptstyle \mathrm{SI}}.k \mapsto h\Big) \end{array} \right\rangle$$

SI-READ-SPEC
$$\langle {\scriptstyle \mathrm{SI}}.k \mapsto_c ov \rangle$$
$$\mathsf{read}\ c\ k$$
$$\langle ov.\ {\scriptstyle \mathrm{SI}}.k \mapsto_c ov \rangle$$

Fig. 10. Specification for snapshot isolation (SISpecs).

Start. In snapshot isolation, when starting a transaction, we take a snapshot of the database. It is on the basis of this snapshot that the sequential reasoning inside the transactions happens: If the transaction does not include any writes, then all reads happen from this snapshot. In the precondition of SI-START-SPEC, the client will provide its connection state together with the state of the key-value store in form of global points-to resources, which is typically obtained from an invariant. The start operation then changes the connection state to active such that it holds information about its start snapshot (${\scriptstyle \mathrm{SI}}.\mathsf{ConnectionState}(c, \mathsf{Active}(m))$). As the sequential reasoning inside transactions happens based on the start snapshot, all the local points-to resources are created with values equal to the current state of the key-value store. As the start of a transaction does not apply any updates, the global points-to resources are given back without modifications for the client to reestablish its invariant. When the transaction starts modifying the local points-to resources by doing writes, we lose the ability to distinguish the original snapshot from client changes. That is why the SI-START-SPEC rule includes the resource $_{\text{SI}}.\mathsf{KeyUpdStatus}(c, k, \mathsf{False})$ in the postcondition. This resource states, for a particular key, whether the client did an update to this key. When talking about the SI-WRITE-SPEC rule next, we will see how the resource is updated and that it is this point that distinguishes our write specification from a store operation for the heap.

Read and Write. The read and write operations are not concerned with the global points-to resources, as they are working on a snapshot of the database, initially captured at start time in the local-points to resources. Hence, a local points-to resource $_{\text{SI}}.k \mapsto_c vo$ is used in the SI-READ-SPEC rule to obtain a specification that is similar to the standard specification of a load operation on the heap. Likewise, the SI-WRITE-SPEC rule is manipulating the points-to predicate similarly to a store operation for the heap, except the change is recorded in the $_{\text{SI}}.\mathsf{KeyUpdStatus}(c, k, b)$ resource which we need at commit time.

Commit. Before delving into the details of the SI-COMMIT-SPEC rule, we remind the reader how transactions are supposed to commit under snapshot isolation. According to the snapshot isolation commit-check, we have to check if there are any write-conflicts between what the transaction wrote and what other concurrent transactions have done in the meantime. Therefore, to use the SI-COMMIT-SPEC rule, one must provide (1) the update status for each key in the starting snapshot, in the form of $_{\text{SI}}.\mathsf{KeyUpdStatus}(c, k, b)$, together with the updated value from the local points-to resources (mc holds the information for these resources); (2) the starting snapshot of the transaction, in the form of $_{\text{SI}}.\mathsf{ConnectionState}(c, \mathsf{Active}(ms))$, where ms is exactly the snapshot gained from using the SI-START-SPEC rule; (3) the current snapshot of the key-value store, that is, the commit snapshot denoted as m, in the form of global points-to resources for all the keys in the start snapshot. Having provided the necessary resources in the precondition of the SI-COMMIT-SPEC rule, the postcondition expresses that the operation can proceed in one of two ways: Either the transaction unsuccessfully commits and the state of the key-value store remains the same as the current state, or the transaction successfully commits and the state of the key-value store is updated with the updates of the transaction. To express the update to the key-value store we use the function $update\_hist$, which takes as arguments both a pair consisting of value and a boolean, and a history. The history is updated, by appending the value to it, if the update-status is true (the update status being the boolean argument). In either case of the postcondition, evidence is provided as to why the case happened: If the transaction committed successfully, $_{\text{SI}}.can\_commit(m, ms, mc)$ will be true, which expresses that there are no write-conflicts with other transactions. Dually, if the transaction is unsuccessful in committing, $_{\text{SI}}.can\_commit(m, ms, mc)$ is false, entailing at least one write-conflict with another transaction. In Figure 11, we display the predicate for which $_{\text{SI}}.can\_commit$ is a decidable procedure. The predicate captures that if the transaction did an update, ($mc[k] = \mathtt{Some}\ (p, \mathsf{True})$), then the start snapshot and the commit snapshot must be equal ($m[k] = ms[k]$).

$$\mathsf{CanCommit}(m, ms, mc) \triangleq \forall k \in \mathsf{Keys},\ mc[k] = \mathtt{Some}\ (p, \mathsf{True}) \Rightarrow m[k] = ms[k].$$

Fig. 11. The CanCommit predicate for which the *can_commit* function is a decidable procedure.

Observations. This completes our explanation of our specifications of the snapshot isolation operations. As for read uncommitted and read committed, clients are able to make observations using the Seen resource. In the exclusion property (8) and the creation property (9) below, we see how Seen resources work on histories instead of sets of values. Moreover, we see that from the exclusion property, a client is able to conclude that an observed history must be a prefix of the history representing the key-value store state. In Appendix A, we discuss the differences between observations made in read uncommitted, read committed and snapshot isolation.

$$\textsf{RU}.\boxed{\mathsf{GlobalInv}} * \textsf{SI}.\mathsf{Seen}(k, h') * \textsf{SI}.k \mapsto h \vdash \Rrightarrow \textsf{SI}.k \mapsto h * h' \leq h \tag{8}$$

$$\textsf{SI}.\boxed{\mathsf{GlobalInv}} * \textsf{SI}.k \mapsto h \vdash \Rrightarrow \textsf{SI}.k \mapsto h * \textsf{SI}.\mathsf{Seen}(k, h) \tag{9}$$

To conclude the presentation of the snapshot isolation specification, we will a state the complete specification similar to (1) and (5) that formally defines the snapshot specification (SISpec), see (10).

$$\mathsf{SISpec} \triangleq \Rrightarrow \textsf{SI}.\mathsf{InitKVS} * \mathop{\Huge\ast}_{k \in \mathsf{Keys}} \textsf{SI}.k \mapsto [\,] * \mathsf{SISpecs} * \textsf{SI}.\boxed{\mathsf{GlobalInv}} \tag{10}$$

Snapshot Isolation Examples. For snapshot isolation, we have proven three examples based on phenomena from the literature (Adya [1999]; Berenson et al. [1995] and 1992 ANSI SQL standard) and a bank transfer example. The examples based on phenomena are the *write skew* example in Figure 12, the *read skew* example in Figure 13 and the *non-repeatable read* example in Figure 14. The formal proofs of these three examples proceed in a manner similar to the proof shown for read uncommitted. For details see the accompanying Rocq formalization. In addition to the examples for snapshot isolation presented in this section, we have verified a number of additional examples, displaying important properties of snapshot isolation, which can be found in Appendix B.

start	‖	start
read y	‖	read x
write x 1	‖	write y 1
assert(commit)	‖	assert(commit)

Fig. 12. Write skew.

start	‖	start
write x 1	‖	v_x = read x
write y 1	‖	v_y = read y
assert(commit)	‖	assert($v_x = v_y$)
	‖	assert(commit)

Fig. 13. Read skew.

start	‖	start
write x 1	‖	v_1 = read x
assert(commit)	‖	v_2 = read x
	‖	assert($v_1 = v_2$)
	‖	assert(commit)

Fig. 14. Non-repeatable read.

The write skew example asserts that all executions between a transaction that reads from one key and writes to another key and another transaction that does the same, but swaps the keys, will commit. Note that even the execution in which both transactions share the same snapshot, in form of the initial snapshot, succeeds in snapshot isolation whereas it is not allowed under serializability because serializability requires there to be an order amongst transactions. Naturally, proving the assertions in this example relies on the inclusion of the CanCommit predicate in the SI-COMMIT-SPEC rule. The read skew example asserts that a transaction will either see all the writes or no writes from other transactions. Finally, the non-repeatable read example asserts that once a transaction sees a write from another transaction it can not unsee it.

Bank transfer. As opposed to the arguably contrived examples based on phenomena from the literature, the bank transfer example in Figure 15 represents a common use case. Namely, the use case in which an amount of money is transferred from one account to another account in a banking system, given that there is sufficient funds in the source account. The example in Figure 15 contains a single transaction, which we imagine could be part of a larger system, and in comments it is shown how each line of code interacts with the separation logic resources, and, most importantly, how the invariant of the system is updated. The invariant for this example, which could be a part of a larger invariant with multiple accounts, is simply: $\exists h_{src}\ h_{dst}\ v_{src}\ v_{dst},\ \textsc{si}.src \mapsto h_{src} +\!\!+ [v_{src}] * \textsc{si}.dst \mapsto h_{dst} +\!\!+ [v_{dst}]$ (it has been boxed and the existential quantification has been omitted in Figure 15). It states that there are two accounts, named *src* and *dst*, which points to histories with the latest updates being v_{src} and v_{dst} respectively (these are the current amounts in the accounts). Using the invariant and SI-START-SPEC, a snapshot is created in Figure 15 for the duration of the transaction in which the amount in the accounts are v_{src} and v_{dst}. The if-statement is used to check whether the balance in the source account is sufficient to withdraw the amount that will be transferred (we want to avoid a negative balance). In case the balance is sufficiently large, the amount is withdrawn from the source account and inserted into the destination account using the SI-WRITE-SPEC rule. Hence, the starting snapshot (the local points-to resources) is changed to a snapshot in which the accounts contain the updated values $v_{src} - amount$ and $v_{dst} + amount$, respectively. At commit time, it is the updates in this new snapshot that we want to push to the key-value store state, represented by the global points-to resources in the invariant. If the commit is successful, we get to do the updates atomically, i.e., write the updates to both accounts. In case the commit is unsuccessful, it is because there is a conflict with another transaction updating the same accounts, cf. the snapshot isolation commit check captured by the CanCommit predicate in SI-COMMIT-SPEC, resulting in the account balances being described by fresh values.

// $\boxed{\textsc{si}.src \mapsto h_{src} +\!\!+ [v_{src}] * \textsc{si}.dst \mapsto h_{dst} +\!\!+ [v_{dst}]}$

start

// $\textsc{si}.src \mapsto_c v_{src} * \textsc{si}.dst \mapsto_c v_{dst}$

bal_{src} = read *src*

// $\textsc{si}.src \mapsto_c v_{src} * \textsc{si}.dst \mapsto_c v_{dst} \wedge bal_{src} = v_{src}$

If $(bal_{src} \geq amount)$ then

{

// $\textsc{si}.src \mapsto_c v_{src} * \textsc{si}.dst \mapsto_c v_{dst} \wedge$
$bal_{src} = v_{src} \wedge v_{src} \geq amount$

write *src* $(bal_{src} - amount)$

// $\textsc{si}.src \mapsto_c v_{src} - amount * \textsc{si}.dst \mapsto_c v_{dst} \wedge$
$bal_{src} = v_{src} \wedge v_{src} \geq amount$

bal_{dst} = read *dst*

// $\textsc{si}.src \mapsto_c v_{src} - amount * \textsc{si}.dst \mapsto_c v_{dst} \wedge$
$bal_{src} = v_{src} \wedge v_{src} \geq amount \wedge bal_{dst} = v_{dst}$

write *dst* $(bal_{dst} + amount)$

// $\textsc{si}.src \mapsto_c v_{src} - amount * \textsc{si}.dst \mapsto_c v_{dst} + amount$
$\wedge\ bal_{src} = v_{src} \wedge v_{src} \geq amount \wedge bal_{dst} = v_{dst}$

}

commit

// *If* $commit = true$ *and* $v_{src} \geq amount$:

$\boxed{\textsc{si}.src \mapsto h_{src} +\!\!+ [v_{src}] +\!\!+ [v_{src} - amount] * \textsc{si}.dst \mapsto h_{dst} +\!\!+ [v_{dst}] +\!\!+ [v_{dst} + amount]}$

Otherwise :

$\boxed{\textsc{si}.src \mapsto h'_{src} +\!\!+ [v'_{src}] * \textsc{si}.dst \mapsto h'_{dst} +\!\!+ [v'_{dst}]}$

Fig. 15. Bank transfer example. [7]

In Chang et al. [2023], a similar bank transfer example is proven in separation logic using the stronger isolation level serializability. As we are able to prove the bank transfer using snapshot isolation, this example demonstrates why applications do not always have to opt for the strongest isolation levels; there can be a weaker isolation level, which provides sufficient consistency guarantees. Weaker isolation levels always come with the benefit that they enhance the throughput of a system due to the increased concurrency — something which is of importance in a banking system with many customers.

We remark that only the snapshot isolation specification is strong enough to prove the examples in this section; they can not be proven using the read uncommitted or the read committed specifications.

[7] For the sake of presentation, this example hides the use of optionals cf. the specification.

6 Snapshot Isolation Implies Read Committed

We can now formally prove that snapshot isolation implies read committed:

Theorem 6.1 (Snapshot isolation implies read committed). *The specification for snapshot isolation* (10) *implies the specification for read committed* (5)*:*

$$\mathsf{SISpec} \rightarrow \mathsf{RCSpec}.$$

The theorem is proven using the same structure as described in Section 4, and the proof amounts to approximately 800 lines of Rocq proof code. We emphasize that while the proof follows the obvious path, coming up with an encoding of the read committed resources using the snapshot isolation resources that makes the proof work is non-trivial. We will refer to the Rocq formalization for the details of the proof.

7 Verifying that a Multi-Version Concurrency Control Key-Value Store Implements Snapshot Isolation

In this section we show that our implementation of the original multi-version concurrency control algorithm for snapshot isolation [Berenson et al. 1995], as a single-node key-value store in a distributed system, satisfies the specification for snapshot isolation. The section is divided into three main parts each constituting a subsection:

(1) *Implementation*: We give an overview of the original multi-version concurrency control algorithm for snapshot isolation from Berenson et al. [1995] in the context of a distributed system.
(2) *Client Proxy Proof*: We outline the client state and how the proof of the client proxy interacts with the server-side using specifications for remote procedure call handlers. We include an excerpt of our ghost theory, which captures relationships among the separation logic predicates we have defined and used.
(3) *Server-Side Proof*: We go into more detail with proving remote procedure call handler specifications. We include a description of the server side resources and our model capturing key properties of snapshot isolation.

We remark that we do not cover the full details of the proof, and use simplifications, as the whole proof is too large to be included in the paper. For the full details, see the Rocq formalization. Note further that a formal consequence of Theorems 4.1 and 6.1 is that the multi-version concurrency control implementation of snapshot isolation also satisfies the specifications for read committed and read uncommitted.

7.1 Implementation

The data structure of the key-value store implementation is a map from keys to lists of value-timestamp pairs. The value-timestamp pairs correspond to the values written in the past with associated commit times, with the last value being the most recent. The server state, moreover, consists of an integer reference, for generating new timestamps to transactions, and a lock for guarding concurrent access to the key-value map and the timestamp integer. The server implements three remote procedure call handlers, in the form of start, read and commit, which together with client proxy code makes up the start, read and commit operations that clients can use. The write operation has no remote procedure call as it is handled locally at the client proxy. The client proxy state can be in one of two modes: The state is empty, because no transaction is active, or the state contains the cached writes and the start timestamp of the active transaction. Note that at most one transaction can be active at a time per client proxy. For a client to have multiple active transactions, a client connection must be initialized using si-init-client-spec for each transaction.

In fact, the client proxy state we talk about in this section is actually per connection. Upon using the start operation, the client proxy asks the server to provide it with a start timestamp. The server serves this by incrementing its timestamp counter and returning a fresh value. Having started a transaction, the read and write operations can now be invoked. In its implementation of the write operation, all the client proxy does is to record the update in its cache. The read operation is more complicated. First, it is checked whether the key, on which the read operation is invoked, has an update in the client cache. If there is an update, the update is returned, otherwise, the read handler of the server is asked to provide a value for the key. Together with the key, the read handler is called with the start timestamp of the transaction, as the reading must be done according to the start snapshot. The start snapshot is represented by the start timestamp. The server uses the key and starting timestamp to retrieve the correct value by going back into the list of value-timestamp pairs. The value with the largest timestamp strictly less than the starting timestamp is returned. Upon commit time, when the commit operation is invoked on the client proxy, the client proxy invokes the commit handler on the server by providing the start timestamp and all the cached writes of the transaction. The commit handler checks that for all the keys in the cache, no updates have been made after the starting timestamp (this is the snapshot isolation commit check, cf. Figure 11). If the check goes well, all the cached updates are appended to the lists of value-timestamp pairs. The timestamp, in a newly added pair, will be a fresh timestamp from the server state higher than all previous timestamps.

In the following two sections we will provide more detailed explanations of the implementation when needed to describe proof details.

7.2 Client Proxy Proof

The client proxy proof serves as a middle layer between the top level specifications of Figure 10 and the server side and makes use of remote procedure call handler specifications. When a client invokes the start, read, write or commit operation, it is the client proxy that gets called. Therefore, it is the client proxy who receives the resources in the precondition of the specifications, and it is also the client proxy who is ultimately responsible for returning the resources in the postcondition.

Client-Side Resources. Upon gaining the resources in the precondition of the specifications in Figure 10, the client proxy takes a lock guarding its state and resources. The lock is used to prevent undefined behavior if the client proxy is used concurrently by multiple threads of the client. The lock is implemented as a spin lock, similarly to what is presented in Birkedal and Bizjak [2017, Section 8.6]. The essence of the specifications for the acquire and release operations of the lock is that one gets to have ownership of some resources protected by the lock. Acquire takes ownership of the resources, and release gives back the ownership to the lock for others to take. The structure of the resources protected by the client proxy lock is displayed (11) ("$\ldots$" is notation for resources which we have omitted for presentation purposes). The resources consist of a pointer ℓ, which points to the client state. The client state is either active, pointing to a starting snapshot t and a client cache *cache*, or the state is in the inactive mode and points to nothing. It is by using the client proxy resources in the lock, together with the resources from the preconditions of the top level specifications that the client proxy gathers the necessary resources for communicating with the server.

$$Lock_{client}(\ell) \triangleq \exists\, state.\ \ell \hookrightarrow state * \big((state = \texttt{Some}\ (t, cache) * \ldots) \lor (state = \texttt{None} * \ldots)\big) \qquad (11)$$

Network Communication. The network model of Aneris is an unreliable network based on UDP entailing that messages can be dropped, duplicated and reordered. We have abstracted away from dealing with the low-level details of an unreliable network by utilizing the remote procedure call library of Gondelman et al. [2023], which implements reliable communication channels on top of the unreliable network. Hence we can think of the client-server communication as the client invoking handlers provided by the server. The interaction between the client and the server in the proof will therefore be happening through the specification of the handlers. Specifications corresponding to SI-INIT-CLIENT-SPEC and SI-INIT-KVS-SPEC come directly, for free, from the reliable communication library, and the write operation does not communicate with the server. Therefore, we have provided handlers for the start, read and commit operations. In Figure 16, we present the specification for the read handler. The read handler is the simplest of the three handlers, as the start and commit handler has preconditions consisting of multiple non-trivial view shifts (formally $P \twoheadrightarrow \Rrightarrow Q$ is view shift from P to Q). View shifts in the precondition of handlers forces the client proxy to prove that given resources P, the server can update P to the resources Q. To explain the read handler specification, we first introduce our ghost theory.

HT-READ-HANDLER

$$\left\{\mathsf{TimeFrag}(t) * \mathsf{SnapFrag}(t, M_t) * M_t[k] = \mathtt{Some}\ H\right\}$$
$$\quad read\_handler\ k\ t$$
$$\left\{ov.\ \left(ov = \mathtt{None} * H = [\,]\right) \vee \left(\exists\, v, t'.\ ov = \mathtt{Some}\ v * last(H) = (v, t')\right)\right\}^{srv}$$

Fig. 16. Read handler specification.

Ghost Theory. Similar to the proof sketch in Section 4, we will be working with a ghost theory of separation logic predicates together with rules that relate the different predicates. Our ghost theory consists, among others, of the following resources: $\mathsf{TimeGlobal}(t)$, $\mathsf{TimeLocal}(t)$, and $\mathsf{TimeFrag}(t)$, all parameterized by a timestamp t; $\mathsf{SnapFrag}(t, M_t)$ and $\mathsf{SnapAuth}(S)$ parameterized by a timestamp t and a map from keys to histories M_t and a map S from timestamps to maps from keys to histories; $\mathsf{MemGlobal}(M)$ and $\mathsf{MemLocal}(M)$, parameterized by a map M from keys to histories; and $k \mapsto H$ parameterized by a key k and a history H. Note that in this section histories are different from the terminology used in Section 5. In this section, histories are lists of value-timestamp pairs as opposed to lists of values (in the client-facing specs in Section 5 there is no need to use timestamps). The resource $k \mapsto H$ is also different from the key-value store points-to resources in Section 5, but it is ultimately $k \mapsto H$ that makes up the key-value store points-to resources we have seen in Section 5. Therefore, in this section we will think of $k \mapsto H$ as the key-value store points-to resources used in the SI-START-SPEC and SI-COMMIT-SPEC rules. An excerpt of the laws of our ghost theory are shown in Figure 17. Going forward, we will go into details with our ghost theory laws as needed for explaining proof details. Let us now return our attention back to the read handler. The precondition

$$\mathsf{TimeGlobal}(t) * \mathsf{TimeLocal}(t) * t < t' \vdash \mathsf{TimeGlobal}(t') * \mathsf{TimeLocal}(t') * \mathsf{TimeFrag}(t') \tag{Law 1}$$
$$\mathsf{TimeGlobal}(t) * \mathsf{TimeLocal}(t') \vdash \mathsf{TimeGlobal}(t) * \mathsf{TimeLocal}(t') * t = t' \tag{Law 2}$$
$$\mathsf{MemGlobal}(M) * \mathsf{MemLocal}(M') \vdash \mathsf{MemGlobal}(M) * \mathsf{MemLocal}(M') * M = M' \tag{Law 3}$$
$$k \mapsto H' * \mathsf{MemGlobal}(M) * \mathsf{MemLocal}(M) \vdash k \mapsto H * \mathsf{MemGlobal}(M[k \mapsto H]) * \mathsf{MemLocal}(M[k \mapsto H]) \tag{Law 4}$$
$$\mathsf{SnapFrag}(t, M_t) * \mathsf{SnapAuth}(S) \vdash \mathsf{SnapAuth}(S) * S[t] = M_t \tag{Law 5}$$
$$\mathsf{SnapAuth}(S) * t \notin \mathrm{dom}(S) \vdash \mathsf{SnapAuth}(S[t \mapsto M]) * \mathsf{SnapFrag}(t, M) \tag{Law 6}$$

Fig. 17. Excerpt of ghost theory laws.

includes some client state: The transaction has a starting timestamp t with associated starting snapshot M_t. Furthermore, the client knows that the key k has the history H in this snapshot. The postcondition expresses that the read handler must return a value in accordance with the history H.

Next, we turn our attention to the server side to, among other things, see how the read-handler specification is proven.

7.3 Server-Side Proof

The server-side proof is exactly the proof that the three remote procedure call handlers (start, read and commit) implement their specifications. As soon as one of the handlers gets invoked, they will have to acquire the server lock before proceeding.

Server-Side Resources. The server lock resources are displayed (12). Similarly to the client lock, the server lock has memory pointers to local state: ℓ_{kvs} points the key-value store itself (kvs maps keys to lists of value-timestamp pairs) while ℓ_t points to the integer/timestamp variable used to generate start and commit timestamps. Furthermore, the lock resources includes the timestamp resource $\mathsf{TimeLocal}(t)$ and the memory resource $\mathsf{MemLocal}(M)$. Here $\mathsf{isMap}(kvs, M)$ asserts that the ghost map M, defined in the logic, corresponds to kvs, which represent the key-value store in the program. Lastly, we assert that our "model" holds, i.e., $\mathsf{Model}(M, S, t)$ for the snapshots S (S is a map from timestamps to maps from keys to histories).

$$\begin{gathered}Lock_{server}(\ell_{kvs}, \ell_t) \triangleq \exists\, kvs, t, m, M, S. \ell_{kvs} \hookrightarrow kvs * \ell_t \hookrightarrow t * \mathsf{TimeLocal}(t) * \mathsf{MemLocal}(M) * \\ \mathsf{isMap}(kvs, M) * \mathsf{Model}(M, S, t)\end{gathered} \tag{12}$$

Mathematical Model. As part of our verification effort, we have defined a mathematical model to reflect key properties of snapshot isolation in the state of the system. The state of the system is here captured by the logical representation of the key-value store M, all the snapshots captured in S and the latest timestamp t. The model does not own any separation logic resources, it consists purely of logical facts (i.e., the $\mathsf{Model}(M, S, t)$ predicate is a persistent predicate). An important property of our model is that it defines a notion of a *cut* for the state, see the definition (13).

$$Cut(t, H_k^t, H_k) \triangleq \exists H'_k,\ H_k = H_k^t +\!\!+\, H'_k \wedge \left(\forall\, (v, t') \in H_k^t,\ t' < t\right) \wedge \left(\forall\, (v, t') \in H'_k,\ t < t'\right) \tag{13}$$

A cut captures that a history H_k can be split in two parts according to a timestamp t, such that all the values in the first part (H_k^t) have timestamps strictly less than t, and all the values in the latter part (H'_k) have timestamps strictly greater than t. For our model, it holds that if the timestamp t is associated with a snapshot of the key-value store M_t, and H_k^t is the history in the snapshot for the key k, while H_k is the current history for the key, then t will cut H_k and the first part of the cut be H_k^t. We remark that only start timestamps have cuts, as all the timestamps in histories are commit timestamps because values are written at commit time (otherwise the strict inequalities in a cut would not make sense). Likewise, we will only associate starting timestamps with a snapshot of the key-value store in our proof development (the domain of S only has starting timestamps).

$$\mathsf{Model}(M, S, t) \vdash \left(S[t'] = M_{t'} \wedge M_{t'}[k] = H_k^{t'} \wedge M[k] = H_k\right) \Rightarrow Cut(t', H_k^{t'}, H_k) \tag{14}$$

This cut property of the model is formally defined (14), and it is in fact this property the proof of the read handler relies on.

Read Handler. The code for the read handler in Figure 18 goes through the list of value-timestamp pairs, starting with the most recent for the specified key k, until it finds a value that has a timestamp strictly less than the argument timestamp t. In accordance with the read handler specification in Figure 16, we must show that this value is indeed the last value of H if H is non-empty. Specifically, at this point in the proof we know that there exists some H_1 and H_2 s.t. H_k (the current history of key k) is composed by H_1 and H_2 in the following way: $H_k = H_1 +\!\!+\, H_2$. Furthermore, we can

```
let start_handler time =
  let timeNext = !time + 1 in
  time := timeNext;
  timeNext

let rec get_last opList sTime =
  match opList with
  | None → None
  | Some list →
    let ((v, time), tail) = list in
    if time = sTime then assert false
    else if time < sTime then Some v
    else get_last tail

let kvs_get_last timeKeyPair kvs =
  let (k, sTime) = timeKeyPair in
  get_last (kvs_get k kvs) sTime

let read_handler kvs timeKeyPair =
  kvs_get_last timeKeyPair !kvs
```

```
let commit_handler kvs cacheData time =
  let cTime = !time + 1 in
  let (sTime, cache) = cacheData in
  if map_is_empty cache then true
  else
    let b = map_forall
        (fun k _ →
          let opList = (map_lookup k !kvs) in
          let list =
            if opList = None then list_nil
            else unSOME opList
          in
          check_key k sTime list) cache
    in
    if b then
      time := cTime;
      kvs := update_kvs !kvs cache cTime;
      true
    else false
```

Fig. 18. Excerpt of server-side handling code.

conclude that $\forall\, v, t'.\ (v, t') \in H_1 \implies t' < t$ and $\forall\, v, t'.\ (v, t') \in H_2 \implies t < t'$. Notice how this entails that t cuts H_k with regard to H_1, that is, $Cut(t, H_1, H_k)$ holds. As it is the value of H_1 which will be returned, the proof of the read handler specification boils down to showing that $H = H_1$. To reach our goal, we must use a resource from the global invariant, shown in (15).

$$\boxed{\mathsf{GlobalInv}} \triangleq \exists t_G, M_G, S_G. \mathsf{TimeGlobal}(t_G) * \mathsf{MemGlobal}(M_G) * \mathsf{SnapAuth}(S_G) * \mathsf{Model}(M_G, S_G, t_G) \tag{15}$$

We have defined the global invariant to contain the resources $\mathsf{TimeGlobal}(t_G)$, $\mathsf{MemGlobal}(M_G)$, $\mathsf{SnapAuth}(S_G)$ and the logical facts of the model $\mathsf{Model}(M_G, S_G, t_G)$. Having $\mathsf{SnapFrag}(t, M_t)$ in the read handler precondition, we can use $\mathsf{SnapAuth}(S_G)$ from the global invariant together with Law 5 of our ghost theory (Figure 17) to conclude $S_G[t] = M_t$. As we know $M[k] = H_k$ (H_k is the current history of k), we can use $\mathsf{MemGlobal}(M_G)$ from the global invariant and $\mathsf{MemLocal}(M)$ from the server lock together with Law 3 of our ghost theory to obtain the equality $M_G[k] = H_k$. This gives us the logical facts needed to conclude that t cuts H_k with regard to H $\left(Cut(t, H, H_k)\right)$ using the cut property of the model (14). Thus, we have established two cuts $Cut(t, H_1, H_k)$ and $Cut(t, H, H_k)$. But, if a timestamp cuts the same history with regard to two different histories, then the two histories must be equal, giving us $H = H_1$, as needed.

Start Handler. The job of the start handler in Figure 18 is to hand out a fresh timestamp for starting transactions. This is done using the timestamp reference of the server state, which is safe as the server lock is held upon using the start handler. Using $\mathsf{TimeGlobal}(t_G)$ from the global invariant and $\mathsf{TimeLocal}(t)$ from the server lock, together with Laws 1 and 2 of our ghost theory, the timestamp is updated to match the server state. Moreover, $\mathsf{SnapAuth}(S_G)$ in the global invariant is updated using Law 6 of the ghost theory, with $\mathsf{SnapFrag}(t, S_G)$ being given to the client proxy, such that the newly generated start timestamp points to the correct snapshot and $\mathsf{SnapFrag}(t, S_G)$ can be used by the client with Ht-read-handler later. To satisfy the model constraints in the global invariant and the server lock upon returning, we must update our model. Figure 19 defines two valid updates to the model, in this case we will be using property 1.

$$\mathsf{Model}(M, S, t) * t < t' \vdash \mathsf{Model}(M, S[t' \mapsto M], t') \qquad \text{(Property 1)}$$

$$\mathsf{Model}(M, S, t) * t < t' \vdash \mathsf{Model}(\mathit{update\_kvs}(M, C, t'), S, t') \qquad \text{(Property 2)}$$

Fig. 19. Model update properties.

Commit Handler. The code of the commit handler Figure 18 essentially consists of two parts. First, it is checked whether the key-value store can be updated in a sound way according to the snapshot isolation commit-check. The call to *check_key* is the snapshot isolation commit-check for a single key. The *check_key* function takes the starting timestamp of the transaction as an argument and tests whether there is a more recent write from another transaction in the history of the argument key, after the start timestamp. It is this part of the code which generates the *can_commit* status in the SI-COMMIT-SPEC rule. Given that the commit-check went well, the second part of the code is responsible for updating the key-value store with the writes of the transaction. Now, the server state must be updated in a way that is sound with regard to our model. Luckily, model-update property 2 in Figure 19 imitates the changes of the *update_kvs* operation used to update the key-value store. When the *update_kvs* operation updates *kvs* using the memory pointer in the server lock (12), the resource $\mathsf{MemLocal}(M)$ together with $\mathsf{MemGlobal}(M_G)$ from the global invariant and the key-value store points-to resources from the precondition of the SI-COMMIT-SPEC rule, can be updated to match the updated key-value store. This is done using Laws 3 and 4 of our ghost theory. Moreover, the timestamp resources in the global invariant and the server lock are updated using the same approach as in the start handler. As all resources are updated in a way that satisfies our model, the global invariant can be closed, and the server lock is re-established and can be released upon returning.

This completes our overview of the implementation and its proof; for more details, see the accompanying Rocq formalization.

8 Related Work

Transactions in Separation Logic. While this paper presents the first separation logic specifications for weak isolation levels, there has been earlier work on separation logic specification and verification of transactions achieving strict serializability [Papadimitriou 1979]. Strict serializability is serializability (transactions appear to execute one at a time) but with the added requirement that the ordering of committed transactions respect real time order. GoTxn [Chajed et al. 2022] is a verified transaction library, which uses two-phase commit locking for concurrency control and it is verified in the Perennial separation logic [Chajed et al. 2019] (which, like Aneris, is also defined on top of Iris). GoTxn stores data on disk in a non-distributed setting and is implemented on top of GoJournal [Chajed et al. 2021] which provides crash safety. Likewise, the vMVCC [Chang et al. 2023] transaction library uses multi-version concurrency control to implement a more advanced form of concurrency control than GoTxn's approach. Chang et al. [2023] implement an in-memory key-value store in a non-distributed setting, and specify and verify it in Rocq using the Perennial separation logic, although Chang et al. [2023] do not use Perennial's support for reasoning about durable storage and crash safety.

Naturally, specifications for (strict) serializability and weak isolation levels are vastly different. To compare with the strongest isolation level we have specified, i.e., snapshot isolation, we note that a specification for snapshot isolation must view the state of the database twice, once to see the starting snapshot on which reads are made, and once to see the commit snapshot on which updates are committed (technically, this is reflected in our SI-START-SPEC and SI-COMMIT-SPEC, where both of the preconditions include global points-to resources). In contrast, a specification for serializability must only view the state of the database once, as a transaction reads and performs updates on the same snapshot (informally, serializability can be viewed as a special case of snapshot isolation where the start snapshot and the commit snapshot are the same). This difference becomes even more apparent if one tries to specify a higher-order run method, which wraps start and commit operations around a body of transaction code. Indeed, the specification for run based on serializability in Chang et al. [2023] is more concise than the run specification for snapshot isolation we have proven in Appendix A, due to the relative simplicity of serializability (In Appendix A we also include a discussion of why a run specification for read committed and read uncommitted would be impractical.) Contrary to what might seem intuitive, and what one can see in more theoretically inclined database papers, serializability does not always imply snapshot isolation. It depends on which notion of implication is being used. As we explain in Appendix C, the implication between the separation logic specification of serializability and snapshot isolation is not provable.

In contrast to our specification of snapshot isolation, neither vMVCC nor GoTxn provides information about why a commit has not succeeded, i.e., a resource in the style of the one expressed using the CanCommit predicate in the postcondition of si-commit-spec. This means that assertions about commit events can not be proven using the vMVCC or GoTxn specification.

Closing the Gap Between High Level Specifications and Implementations. Other work has also aimed at closing the gap between the seemingly high-level properties of isolation levels and specifications for runnable implementations. Cerone et al. [2015] gives operational specifications, closer to specifications of actual implementations, for a number of isolation levels, and prove that these are equivalent to specifications in the style of weak memory models. Crooks et al. [2017] has developed a state-based model for reasoning about transactional consistency which has been embedded in TLA$^+$[Soethout et al. 2020]. TLA$^+$[Lamport 1992] is a specification language and model checker. A version of its specification language, PlusCal, resembles mainstream programming languages but still has to undergo manual translation to executable code. Raad et al. [2019] shows how a pseudo-code implementation of a local transactional library can implement an axiomatized model of snapshot isolation equivalent to the model of Cerone and Gotsman [2018]. The approach is not based on a program logic or mechanized, and can not reason about client programs. Lesani et al. [2022] lifts linearizable data structures to serializable transactional memory objects using interaction trees as program representations in a mechanized Rocq framework. There is also a line of work on checking database implementations against transaction workloads [Biswas et al. 2021; Bouajjani et al. 2023; Liu et al. 2024; Zhang et al. 2023]. While checkers have shown to be effective at discovering bugs, they can not show the absence of bugs.

Verification of Distributed Systems Using Separation Logic. Mechanized verification of distributed systems using higher-order separation logics have been carried out a number of times with other foci than transactions. For example, Aneris has been used to verify distributed algorithms with high-availability and lower consistency guarantees, such as conflict-free replicated data types, [Nieto et al. 2023, 2022] and a causally-consistent replicated key-value store [Gondelman et al. 2021]. Grove [Sharma et al. 2023] is another higher-order logic for distributed systems based on Perennial with support for reasoning about durable state and crashes in which a replicated key-value store has been verified.

Weak Memory. The area of weak memory is another area in which separation logic has been successfully used to formalize relaxed guarantees, see [Kaiser et al. 2017; Turon et al. 2014; Vafeiadis and Narayan 2013] with Kaiser et al. [2017] being an Iris based logic. While the guarantees of weak isolation for database transactions do not have a direct relationship with the guarantees in weak memory, both areas exhibit very relaxed behavior that makes them difficult to reason about. In the weak memory world, one starts with an operational semantics for the memory, whereas in the work of this paper we show how an implementation of a weak isolation level, in the form of a database, can satisfy our specification.

9 Conclusion and Future Work

In this paper, we have established that separation logic can be used to give modular specifications of and reason about consistency guarantees for transactions. As this work has laid the theoretical foundation for reasoning about transactions in separation logic, there are two obvious paths for extending this work.

First, this work can be seen as a stepping stone to relating specifications for executable code, i.e., the separation logic specifications presented here, to well-established database theory models for reasoning about transactional consistency, i.e., models such as the ones mentioned in Section 1, which define transactional consistency using, for instance, high level operational semantics or dependency graphs. We remark that the per-operation separation logic specifications presented in this paper already do make a connection to existing database theory, since we have shown how they can be used to exclude the phenomena used to define isolation levels in the literature [Adya 1999; Berenson et al. 1995] and 1992 ANSI SQL standard, which for many years was, and by the SQL Standard authors still is, the de-facto standard. Making a stronger connection to one of the mentioned models, e.g., by proving formally that our separation logic specifications relate precisely to dependency graphs, is highly non-trivial. To create such a formal connection, it is necessary to use an extra layer of logical machinery, not part of the standard instantiation of Iris, such as Trillium [Timany et al. 2024], for establishing refinements of state transition systems, or free theorems [Birkedal et al. 2021], for stating invariants on traces created by emitting tags in "ghost-code".

Second, it would be interesting to see the theory applied to various transaction systems. While the database we have implemented is indeed the first executable database that verifiably implements a weak isolation level, and uses the multi-version concurrency control algorithm of [Berenson et al. 1995], modern databases comes with many features; e.g., distributed transactions [Corbett et al. 2013; Garcia-Molina and Salem 1987; Lampson and Sturgis 1981], where data is scattered across a number of different databases. Further, it would be interesting to see the developed theory applied to databases that are resilient to crash failures. This would necessitate using a distributed program logic with support for writing to durable storage and reasoning about crashes, such as Grove [Sharma et al. 2023].

Data Availability Statement

The Rocq formalization and OCaml code accompanying this work is available on Zenodo [Mathiasen et al. 2025].

Acknowledgments

This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation. This work was co-funded by the European Union (ERC, CHORDS, 101096090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

A Utility Code

In addition to the operations of the key-value store API, we have implemented utility operations wait and run. The wait and run operations are implemented on top of the key-value store operations and proven correct with regard to their specifications (the specifications are presented in this section) using the specifications from Figures 3, 5 and 10. The run is similar to the one presented for serializability in Chang et al. [2023], but has a vastly different specification as discussed in Section 9. As we will see in the following, while the utility operations provide new functionality, they are good at showcasing the differences between the isolation levels.

```
let wait cst cond k =
  let rec aux () =
    start cst ;
    match read cst k with
    | None →
      commit cst; aux ()
    | Some v →
      if cond v
      then (commit cst)
      else (commit cst; aux
 ())
in aux ()
```

```
let weak_wait cst cond k =
  start cst;
  let rec aux () =
    match read cst k with
    | None → aux ()
    | Some v →
      if cond v
      then (commitU cst)
      else (aux ())
  in aux ()
```

```
let run cst handler =
  start cst;
  handler cst;
  commit cst
```

Fig. 20. Utility code.

Wait operation. The wait operation comes in two versions: a *weak* version, for read uncommitted and read committed, and a non-weak version for snapshot isolation. The operation works much like a memory fence: It takes as arguments a connection, key and condition. When the operation returns, the condition holds for the value pointed to by the key. The wait operation can be used to synchronize transactions across clients to impose a particular order. The wait and weak-wait are implemented as recursive functions, see Figure 20. Starting with the non-weak wait operation: Upon every call, a new transaction is started to obtain the current snapshot of the key-value store. The snapshot is then checked for the condition by reading from the specified key and seeing whether the condition holds for the value pointed to be the key. If the condition is true, the operation commits the transaction and terminates, otherwise the transaction is committed, and a recursive call is made to check the condition on a more recent snapshot. Notice how every call starts and commits a transaction. Had the wait operation only started one transaction and done the recursive calls inside this transaction, it would never get a new start snapshot and hence not observe changes done by other transactions, including changes that could make the condition true. Now, this part is what distinguishes the non-weak wait from the weak wait: The weak wait operation does not start a new transaction for every recursive call to check the state of the key. This is not needed for read uncommitted and read committed as transactions can see updates made by other transactions during their own execution. Do note that the key-value store does not *have* to show updates to transactions that are already execution, e.g., the key-value store we have implemented in Section 7 have realized the specifications for read uncommitted and read committed through the implications proofs without propagating new values to active transactions, but if you do not, then you might as well implement a stronger isolation level. The specification for the non-weak wait operation is shown in Figure 21. It has three resources in its precondition. First, one must provide the ability to start a transaction in the form of ConnectionState(c, CanStart). Second, the user must show that the provided condition is a test for equality for a specific value v. Third, the user must provide the ability to look at the state of the key-value store all the possibly many times the wait operation will be opening and closing transactions to check for the condition. This is done in the form of an update modality and a view shift ($P \Rrightarrow\!\!\ast\; Q$ is a view shift and is notation for $P \mathrel{-\!\!\ast} {\mid\!\Rrightarrow} Q$) [Jung et al. 2015] under the persistence modality (going forward we will refer to both $\Rrightarrow\!\!\ast$ and ${\mid\!\Rrightarrow}$ as view shifts). We recap that the update modality ${}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2} P$ states that given that the invariants in $\mathcal{E}_1$ are

WAIT-SPEC
$$\left\{\begin{array}{l}\textsf{SI}.\mathsf{ConnectionState}(c, \mathsf{CanStart}) \ast \\ \forall v'. \{\mathsf{True}\}\ \mathsf{cond}\ v'\ \{b.\, b \Rightarrow v = v'\} \ast \\ \Box\ {}^{\top}{\Rrightarrow}^{\mathcal{E}} \left(\exists h,\ \textsf{SI}.k \mapsto h \ast \triangleright (\textsf{SI}.k \mapsto h\ {}^{\mathcal{E}}{\Rrightarrow\!\!\ast}^{\top}\ \mathsf{True})\right)\end{array}\right\}$$
$$\mathsf{wait}\ c\ k\ \mathsf{cond}$$
$$\{().\ \textsf{SI}.\mathsf{ConnectionState}(c, \mathsf{CanStart}) \ast \textsf{SI}.\mathsf{Seen}(k, h +\!\!+ [v])\}$$

WEAK-WAIT-SPEC
$$\left\{\begin{array}{l}\textsf{RC}.\mathsf{ConnectionState}(c, \mathsf{CanStart}) \ast \\ \forall v'. \{\mathsf{True}\}\ \mathsf{cond}\ v'\ \{b.\, b \Rightarrow v = v'\} \ast \\ \Box\ {}^{\top}{\Rrightarrow}^{\mathcal{E}} \left(\exists V,\ \textsf{RC}.k \mapsto V \ast \triangleright (\textsf{RC}.k \mapsto V\ {}^{\mathcal{E}}{\Rrightarrow\!\!\ast}^{\top}\ \mathsf{True})\right)\end{array}\right\}$$
$$\mathsf{wait}\ c\ k\ \mathsf{cond}$$
$$\{().\ \textsf{RC}.\mathsf{ConnectionState}(c, \mathsf{CanStart}) \ast \textsf{RC}.\mathsf{Seen}(k, V \cup v)\}$$

Fig. 21. Wait specification and weak wait specification.

active ($\mathcal{E}_1$ is a set of invariant names commonly referred to as a mask, we omitted invariant names, used as a mean to identify each invariant, in the main body of the paper), updates can be made to satisfy P and establish the invariants in $\mathcal{E}_2$. If the two masks $\mathcal{E}_1$ and $\mathcal{E}_2$ are equal, we can choose to write the update modality as ${\Rrightarrow}_{\mathcal{E}}$ or simply ignore the mask $\mathcal{E}$. The persistence modality ensures that the view shift can be reused all the possibly many times the wait operation will be opening and closing transactions. Specifically, the view shifts allow us, possibly by opening an invariant, to get the key-value store points-to resource for the argument key and later close any invariant using the same key-value store points-to resource. Given the mentioned preconditions, if the wait operation terminates the user obtains evidence of the observation in the form of $\textsf{SI}.\mathsf{Seen}(k, h +\!\!+ [v])$. One can then use the snapshot isolation version of exclusion property (2) (see Section 5) to exclude future key-value store states where the value v has not been written to the key k yet. We remark that in our Rocq development, a more general specification that does not restrict the condition to be an equality is provided. The specification of the weak wait operation is much similar to its stronger counterpart. Figure 21 displays the weak wait specification for read committed, we remark that read uncommitted has an equivalent specification (all resources should be prefixed with $\textsf{RU}.$ instead of $\textsf{RC}.$). The difference between the weak wait operation and the non-weak operation, on the specification level, lies in the kind of observations one gains when the operation returns. The read committed (and read uncommitted observations) $\textsf{RC}.\mathsf{Seen}(k, V \cup v)$ are on sets which imposes no order compared to the histories of snapshot isolation. This only lets the user exclude states based on whether a given write have happened or not but not based on the order it happened.

Using the non-weak wait specification for snapshot isolation, we have proven two examples for concurrently executing transactions that make assertions based on the observations gained from using wait. The first example, *Capturing causality* in Figure 22, uses wait to capture a causality relation between transactions. The second example, *Sequential writes commit* in Figure 23, asserts that committing will be successful as, even though the two transactions write to the same key, there can not be a write-conflict because the second transaction is waiting for the first transaction to finish.

$$\begin{array}{l||l||l}
\mathsf{start} & \mathsf{wait}\ x\ 1 & \mathsf{wait}\ y\ 1 \\
\mathsf{write}\ x\ 1 & \mathsf{start} & \mathsf{start} \\
\mathsf{commit} & \mathsf{write}\ y\ 1 & v_x = \mathsf{read}\ x \\
 & \mathsf{commit} & \mathsf{assert}(v_x = 1) \\
 & & \mathsf{commit}
\end{array}$$

$$\mathit{Inv} \triangleq \exists h_x\ h_y,\ \textsf{SI}.x \mapsto h_x \ast \textsf{SI}.y \mapsto h_y \ast$$
$$((h_x = [\,] \ast h_y = [\,]) \lor (\mathit{last}(h_x) = \mathsf{Some}\ 1))$$

Fig. 22. Capturing causality.

$$\begin{array}{l||l}
 & \mathsf{wait}\ x\ 1 \\
\mathsf{start} & \mathsf{start} \\
\mathsf{write}\ x\ 1 & \mathsf{write}\ x\ 2 \\
\mathsf{assert}(\mathsf{commit}) & \mathsf{assert}(\mathsf{commit})
\end{array}$$

$$\mathit{Inv} \triangleq \exists h,\ \textsf{SI}.x \mapsto h \ast$$
$$((h = [\,]) \lor (h = [1] \ast \mathsf{T}_1) \lor (h = [1; 2] \ast \mathsf{T}_1 \ast \mathsf{T}_2))$$

Fig. 23. Sequential writes commit.

Run operation. The run operation is a higher-order function, which wraps start and commit operations around a body of transaction code. Its specification, shown in Figure 24, naturally follows from the specifications in Figure 10: To be able to start a transaction, the Run-spec rule must have ConnectionState(c, CanStart) and a snapshot of the key-value store. The view shifts used for

Run-spec

$$
\begin{array}{l}
\mathsf{SI}.\mathsf{ConnectionState}(c, \mathsf{CanStart}) \ast \\
{}^{\top}\!\Rrightarrow^{\mathcal{E}} \Big(\exists m,\ P(m) \ast \mathop{\Large\ast}_{(k,h)\in m} \mathsf{SI}.k \mapsto h \ast \triangleright \big(\mathop{\Large\ast}_{(k,h)\in m} \mathsf{SI}.k \mapsto h \ {}^{\mathcal{E}}\!\!\Rrightarrow\!\!\ast^{\top}\ \mathsf{True}\big)\Big) \ast \\
\quad \left\{ \mathop{\Large\ast}_{(k,h)\in ms} (\mathsf{SI}.k \mapsto_c \mathit{last}(h) \ast \mathsf{SI}.\mathsf{KeyUpdStatus}(c,k,\mathsf{False})) \ast P(ms) \right\} \\
\qquad \mathit{body}\ c \\
\forall ms. \left\{ \begin{array}{l}
().\ \mathrm{dom}\ ms = \mathrm{dom}\ mc \ast \\
\quad \mathop{\Large\ast}_{(k,(ov,b))\in mc} (\mathsf{SI}.k \mapsto_c ov \ast \mathsf{SI}.\mathsf{KeyUpdStatus}(c,k,b)) \ast \\
{}^{\top}\!\Rrightarrow^{\mathcal{E}} \Big(\exists m,\ Q(m, ms, mc) \ast \mathop{\Large\ast}_{(k,h)\in m} \mathsf{SI}.k \mapsto h \ast \mathrm{dom}\ m = \mathrm{dom}\ ms \ast \\
\qquad\qquad \triangleright\big(\big(\mathop{\Large\ast}_{\substack{(k,h)\in m\\(k,p)\in mc}} \mathsf{SI}.k \mapsto \mathit{update\_hist}(p,h) \vee \mathop{\Large\ast}_{(k,h)\in m} \mathsf{SI}.k \mapsto h\big)\ {}^{\mathcal{E}}\!\!\Rrightarrow\!\!\ast^{\top}\ \mathsf{True}\big)\Big)
\end{array} \right\} \\
\quad \mathsf{run}\ c\ \mathit{body} \\
\left\{ \begin{array}{l}
v.\ \mathsf{SI}.\mathsf{ConnectionState}(c, \mathsf{CanStart}) \ast Q(m, ms, mc) \ast \\
\Big(v = \mathsf{True} \ast \mathsf{SI}.\mathit{can\_commit}(m, ms, mc) \ast \mathop{\Large\ast}_{\substack{(k,h)\in m\\(k,p)\in mc}} \mathsf{SI}.\mathsf{Seen}(k, \mathsf{SI}.\mathit{update\_hist}(p,h))\Big) \\
\vee \Big(v = \mathsf{False} \ast \neg \mathsf{SI}.\mathit{can\_commit}(m, ms, mc) \ast \mathop{\Large\ast}_{(k,h)\in m} \mathsf{SI}.\mathsf{Seen}(k,h)\Big)
\end{array} \right\}
\end{array}
$$

Fig. 24. Run specification.

taking a snapshot of the key-value store follows the same idea as in the precondition of the wait specification. The differences are that the persistence modality is not needed, as the view shift is only used once to start the transaction, and also that the snapshot can be described by a user-defined predicate P. Having started the transaction, the run specification will have to reason about the body of the transaction based on the snapshot described by P. As the body of the transaction is provided by the user of the specification, the Run-spec rule includes in its precondition a Hoare triple for the transaction body. The Hoare triple for the transaction body has itself as precondition connection points-to resources and update status resources corresponding to the ones obtained from starting a transaction using the first two resources of the Run-spec rule precondition. From using the transaction body Hoare Triple, the connection points-to and updates status resources potentially gets modified. The changes are captured in the map mc in the postcondition of the triple. For committing the transaction, based on the modifications in mc and the snapshot ms, the Run-spec rule uses the two view shifts it also obtains from using the transaction body Hoare triple. These view shifts describe exactly how one can obtain, possible by opening invariants, a commit snapshot of the key-value store and close any invariants using one of two snapshots: The unmodified commit snapshot corresponding to the unsuccessful commit case, or the commit snapshot with the updates described by mc corresponding to the successful commit case. The postcondition of the Run-spec rule reflects the commit status of the transaction body: The return status of the run operation is attached to *can_commit* expressed using a commit snapshot m, a start snapshot ms and modifications mc. All three maps are related by the user-provided predicate Q, just as in the postcondition of the transaction body. For excluding future key-value store states, Seen predicates are also returned based on the commit status of the transaction body.

While the run specification for snapshot isolation is complex, and most likely it is more convenient to reason about transactions running under snapshot isolation using the per operation specifications of Figure 10, the run specifications for read committed and read uncommitted would be far more complex. As the specifications of read and write access shared key-value store state in read uncommitted and read committed, a run specification must take as argument resources in the form of view shifts that can be used to reason about the access to shared key-value store state for all the operations of the transaction body — not only the start and commit operations as in the run specification for snapshot isolation Figure 24.

B Example Portfolio

In addition to the examples presented in the body of the paper, we have verified a number of additional examples for the strongest isolation level of snapshot isolation displaying important properties of snapshot isolation. All the examples have been proven in Rocq using our specifications in Figure 10 and can be found in the accompanying Rocq files. The invariant is listed together with each example, unless an invariant was not needed. This is contrary to the body of the paper where we sometimes omit invariants for presentation purposes. In the following, we will attach some comments to each of the examples.

The *atomic transactions* example in Figure 25 xpress that transactions are atomic using an assert statement: Either all writes in a transaction are committed, or no writes are committed — it is not possible to read one value from one transaction and another value from the other transaction.

The *convenience of points-to* example in Figure 26 shows the convenience of working with points-to predicates as it allows for modularity: One can make a specification for the function f that only mentions the points-to-predicate for the key it takes as argument.

The *disjoint writes* example in Figure 27 asserts that transactions writing to disjoint sets of keys will commit successfully. This holds as the snapshot isolation commit-check in Figure 11 is only concerned with write-conflicts which can not be created from disjoint writes. Furthermore, the disjoint writes example shows the modularity of the specs: No invariant is needed as the state of the key-value store is expressed using points-to predicates. This allows us to distribute the points-to resources for keys x and y to the left and right transactions respectively.

The *read-only transaction* example in Figure 28 asserts that a read-only transaction will commit successfully. As mentioned before, this holds because the snapshot isolation commit-check is only concerned with write-conflicts.

The last example in Figure 29 displays the difference between snapshot isolation and serializability: Snapshot isolation is a weaker property than serializability as any execution of committed transactions allowed under serializability is also allowed under snapshot isolation. We can think of serializability as a special case of snapshot isolation where the start snapshot and the commit snapshot is the same. Now, this comes with the caveat that snapshot isolation guarantees that all transactions will be reading from a valid snapshot, while serializability gives no guarantees about transactions that do not successfully commit. The implication of this is that under snapshot isolation we can make assertions before a transaction has committed. If we refrain from considering that assertions must be made after successfully committing in serializability, we can not create an example, which is only provable under snapshot isolation as an algorithm implementing snapshot isolation can then behave exactly as one implementing serializability. To illustrate this point, observe that the left case in the disjunction of the assertion in Figure 29, $v_x + v_y$ = -2, corresponds to the only execution not allowed under serializability. Namely, the execution in which the writing transactions have the same start snapshot. Had this case not been included in the assertion, we would only have been able to prove the (remaining) assertion for an algorithm implementing serializability, but not for one implementing snapshot isolation.

start	start	start
write x 1	write x 2	v_x = read x
write y 1	write y 2	v_y = read y
commit	commit	assert$(v_x = v_y)$
		assert(commit)

$$Inv \triangleq \exists h,\ \textsc{si}.x \mapsto h * \textsc{si}.y \mapsto h$$

Fig. 25. Atomic transactions.

start	start
write x 1	$r = f(x)$
commit	write y r
	commit

$$Inv \triangleq \exists h,\ \textsc{si}.x \mapsto h$$

Fig. 26. The convenience of points-to.

start	start
write x 1	write y 1
assert(commit)	assert(commit)

Fig. 27. Disjoint writes commit.

start	start
write x 1	read x
commit	assert(commit)

$$Inv \triangleq \exists h,\ \textsc{si}.x \mapsto h$$

Fig. 28. Read-only transactions commit.

start	wait z 1	wait z 1	wait z 1
write x 1	start	start	start
write y 1	v_x = read x	v_y = read y	v_x = read x
write z 1	If $(v_x = 1)$ then	If $(v_y = 1)$ then	v_y = read y
commit	{write y (-1) }	{write x (-1) }	assert$(v_x + v_y = \text{-}2 \lor v_x + v_y \geq 0)$
	commit	commit	commit

$$Inv \triangleq \exists h_x\ h_y\ h_z,\ \textsc{si}.x \mapsto h_x * \textsc{si}.y \mapsto h_y * \textsc{si}.z \mapsto h_z *$$
$$\Big((h_x = [\,]) \lor (\exists\, v_x, v_y.\ last(h_x) = \text{Some } v_x * last(h_y) = \text{Some } v_y * (v_x = 1 \lor v_x = \text{-}1) * (v_y = 1 \lor v_y = \text{-}1))\Big)$$

Fig. 29. Capturing the difference between snapshot isolation and serializability.

C Serializability Does not Imply Snapshot Isolation in Separation Logic

Consider an example of two transactions with a single write operation both writing to the same key. All executions for these transactions are valid under serializeability. For snapshot isolation, only some executions are: the execution in which the two transactions share the same starting snapshot is not. The notion of implication we use (which is the natural one when thinking of specifications in terms of Hoare Triples), requires us to show that any execution valid for serializability is also valid under snapshot isolation. Hence, the implication can not be proven. In other papers, the implication is sometimes interpreted as: if there exists an execution valid for serilizability, then there must exist some execution (for the same set of transaction of course) which is valid under snapshot isolation. Hence, using this definition the implication can be proven.

References

A. Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Technical Report. USA.

A. Adya, B. Liskov, and P. O'Neil. 2000. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073).* 67–78. doi:10.1109/ICDE.2000.839388

Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. *SIGMOD Rec.* 24, 2 (may 1995), 1–10. doi:10.1145/568271.223785

Lars Birkedal and Aleš Bizjak. 2017. Lecture Notes on Iris: Higher-Order Concurrent Separation Log. http://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf. (2017).

Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. 2021. Theorems for Free from Separation Logic Specifications. *Proc. ACM Program. Lang.* 5, ICFP, Article 81 (aug 2021), 29 pages. doi:10.1145/3473586

Ranadeep Biswas, Diptanshu Kakwani, Jyothi Vedurada, Constantin Enea, and Akash Lal. 2021. MonkeyDB: effectively testing correctness under weak isolation levels. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 132 (Oct. 2021), 27 pages. doi:10.1145/3485546

Ahmed Bouajjani, Constantin Enea, and Enrique Román-Calvo. 2023. Dynamic Partial Order Reduction for Checking Correctness against Transaction Isolation Levels. *Proc. ACM Program. Lang.* 7, PLDI, Article 129 (June 2023), 26 pages. doi:10.1145/3591243

Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. 2012. Eventually consistent transactions. In *Proceedings of the 21st European Conference on Programming Languages and Systems* (Tallinn, Estonia) *(ESOP'12)*. Springer-Verlag, Berlin, Heidelberg, 67–86. doi:10.1007/978-3-642-28869-2_4

Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *26th International Conference on Concurrency Theory (CONCUR 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 42)*, Luca Aceto and David de Frutos Escrig (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 58–71. doi:10.4230/LIPIcs.CONCUR.2015.58

Andrea Cerone and Alexey Gotsman. 2018. Analysing Snapshot Isolation. *J. ACM* 65, 2, Article 11 (Jan. 2018), 41 pages. doi:10.1145/3152396

Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying Concurrent, Crash-Safe Systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 243–258. doi:10.1145/3341301.3359632

Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. GoJournal: a verified, concurrent, crash-safe journaling system. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 423–439. https://www.usenix.org/conference/osdi21/presentation/chajed

Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich. 2022. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 447–463. https://www.usenix.org/conference/osdi22/presentation/chajed

Yun-Sheng Chang, Ralf Jung, Upamanyu Sharma, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2023. Verifying vMVCC, a high-performance transaction library using multi-version concurrency control. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 871–886. https://www.usenix.org/conference/osdi23/presentation/chang

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (Aug. 2013), 22 pages. doi:10.1145/2491245

Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Washington, DC, USA) *(PODC '17)*. Association for Computing Machinery, New York, NY, USA, 73–82. doi: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* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 715–726.

Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '87)*. Association for Computing Machinery, New York, NY, USA, 249–259. doi:10.1145/38713.38742

Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. 2021. Distributed causal memory: modular specification and verification in higher-order distributed separation logic. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. doi:10.1145/3434323

Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, and Lars Birkedal. 2023. Verifying Reliable Network Components in a Distributed Separation Logic with Dependent Separation Protocols. *Proc. ACM Program. Lang.* 7, ICFP, Article 217 (aug 2023), 31 pages. doi:10.1145/3607859

Jepsen. 2025. Jepsen Analyses. https://jepsen.io/analyses Accessed 2025-02-21.

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 256–269. doi:10.1145/2951913.2951943

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. doi:10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 637–650. doi:10.1145/2676726.2676980

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 17:1–17:29. doi:10.4230/LIPIcs.ECOOP.2017.17

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 (Dec. 2017), 1–34. doi:10.1145/3158115

Bas Ketsman, Christoph Koch, Frank Neven, and Brecht Vandevoort. 2023. Concurrency control for database theorists. *SIGMOD Rec.* 51, 4 (jan 2023), 6–17. doi:10.1145/3582302.3582304

Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. 336–365. doi:10.1007/978-3-030-44914-8_13

Leslie Lamport. 1992. Hybrid Systems in TLA^+. In *Hybrid Systems*, Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel (Eds.). Lecture Notes in Computer Science, Vol. 736. Springer, 77–102. doi:10.1007/3-540-57318-6_25

Butler W. Lampson and Howard E. Sturgis. 1981. Crash Recovery in a Distributed Data Storage System. https://api.semanticscholar.org/CorpusID:263889622

Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C. Pierce, and Steve Zdancewic. 2022. C4: verified transactional objects. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 80 (April 2022), 31 pages. doi:10.1145/3527324

Si Liu, Long Gu, Hengfeng Wei, and David Basin. 2024. Plume: Efficient and Complete Black-Box Checking of Weak Isolation Levels. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 302 (Oct. 2024), 29 pages. doi:10.1145/3689742

Alnor Alnor Mathiasen, Léon Gondelman, Léon Ducruet, Amin Timany, and Lars Birkedal. 2025. Reasoning about Weak Isolation Levels in Separation Logic — Artifact. https://doi.org/10.5281/zenodo.15626657

MicrosoftSQL. 2025. MicrosoftSQL documentation. https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/sql/snapshot-isolation-in-sql-server Accessed 2025-06-02.

Abel Nieto, Arnaud Daby-Seesaram, Léon Gondelman, Amin Timany, and Lars Birkedal. 2023. Modular Verification of State-Based CRDTs in Separation Logic. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 22:1–22:27. doi:10.4230/LIPIcs.ECOOP.2023.22

Abel Nieto, Léon Gondelman, Alban Reynaud, and Lars Birkedal. 2022. Modular Verification of Op-Based CRDTs in Separation Logic. *Proc. ACM Program. Lang.* OOPSLA (2022). Accepted for publication..

Oracle. 2025. Oracle documentation. https://docs.oracle.com/en/database/other-databases/timesten/22.1/introduction/transaction-isolation.html#GUID-7272EB20-51D6-412C-B25D-58986B3668C8 Accessed 2025-06-02.

Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4 (oct 1979), 631–653. doi:10.1145/322154.322158

PostgeSQL. 2025. PostgeSQL documentation. https://www.postgresql.org/docs/7.2/xact-read-committed.html Accessed 2025-06-02.

Azalea Raad, Ori Lahav, and Viktor Vafeiadis. 2019. *On the Semantics of Snapshot Isolation: 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13–15, 2019, Proceedings*. 1–23. doi:10.1007/978-3-030-11245-5_1

Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. 2023. Grove: A Separation-Logic Library for Verifying Distributed Systems. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 113–129. doi:10.1145/3600006.3613172

Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. 2020. Automated Validation of State-Based Client-Centric Isolation with TLA+. In *Software Engineering and Formal Methods. SEFM 2020 Collocated Workshops: ASYDE, CIFMA, and CoSim-CPS, Amsterdam, The Netherlands, September 14–15, 2020, Revised Selected Papers* (Amsterdam, The Netherlands). Springer-Verlag, Berlin, Heidelberg, 43–57. doi:10.1007/978-3-030-67220-1_4

Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Jonas Kastberg Hinrichsen, Léon Gondelman, Abel Nieto, and Lars Birkedal. 2024. Trillium: Higher-Order Concurrent and Distributed Separation Logic for Intensional Refinement. *Proc. ACM Program. Lang.* 8, POPL (2024), 241–272. doi:10.1145/3632851

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 691–707. doi:10.1145/2660193.2660243

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: a program logic for C11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 867–884. doi:10.1145/2509136.2509532

Shale Xiong. 2020. *Parametric Operational Semantics for Consistency Models*. PhD thesis. Imperial College London.

Jian Zhang, Ye Ji, Shuai Mu, and Cheng Tan. 2023. Viper: A Fast Snapshot Isolation Checker. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 654–671. doi:10.1145/3552326.3567492

Received 2025-02-27; accepted 2025-06-27