Alone Together: Compositional Reasoning for Weak Isolation

Abstract

Serializability is a well-understood correctness criterion that simplifies reasoning about the behaviour of concurrent transactions by ensuring they are *isolated* from each other while they execute. However, enforcing serializable isolation comes at a steep cost in performance because it necessarily restricts opportunities to exploit concurrency even when such opportunities would not violate application-specific invariants. As a result, database systems in practice support, and often encourage, developers to use weaker alternatives. These alternatives break the strong isolation guarantees offered by serializable transactions to permit greater concurrency. Unfortunately, the semantics of weak isolation is poorly understood, and usually explained only informally in terms of low-level implementation artifacts. Consequently, verifying high-level correctness properties in such environments remains a challenging problem.

To address this issue, we present a program logic that enables compositional reasoning about the behaviour of concurrently executing weakly-isolated transactions. Notably, our development is parametric over a transaction's specific isolation semantics, and the consistency guarantees provided by the underlying data store, allowing it to be applicable over a range of concurrency control mechanisms. Case studies and experiments on real-world applications demonstrate the utility of our approach, and provide strong evidence that weakly-isolated transactions can be placed on the same formal footing as their strongly-isolated serializable counterparts.

1. Introduction

Database transactions allow users to group operations on multiple objects into a single logical unit, equipped with a set of four key properties - atomicity, consistency, isolation, and durability (ACID). Concurrency control mechanisms provide specific instantiations of these properties to yield different ACID variants that characterize how and when the effects of concurrently executing transactions become visible to one another. *Serializability* is a particularly well-studied instantiation that imposes strong atomicity and isolation constraints on transaction execution, ensuring that any permissible concurrent schedule yields results equivalent to a serial one in which there is no interleaving of actions from different transactions.

The guarantees provided by serializability do not come for free, however - pessimistic concurrency control methods that enforce serializability, for example, require databases to provide expensive techniques such as two-phase locking that incur overhead to deal with deadlocks, rollbacks, and re-execution [23, 28]. Similar criticisms apply to optimistic multi-version concurrency control methods that must deal with timestamp and version management [15]. Regardless of the particular concurrency control method employed, enforcing serializability in a replicated data store additionally requires global coordination among geo-distributed replicas, which makes the store *unavailable* in the presence of network partitions [5, 14, 21, 29].

The tension between serializable transactions, which are easy to reason about but difficult to implement, and more pragmatic variants that are driven by performance and availability considerations, has motivated the development of weaker forms of transaction isolation, beginning as early as 1976 [31]. The ANSI SQL 92 standard defines three such weak isolation levels which are now implemented in many relational and NoSQL databases. Not surprisingly, weakly-isolated transactions have been found to significantly outperform serializable transactions on benchmark suites, both on single-node databases and multi-node replicated stores [5, 8, 45], leading to their overwhelming adoption. A 2013 study [6] of 18 popular ACID and "NewSQL" databases found that only three of them offer serializability by default, and half, including Oracle 11g, do not offer it at all. A 2015 study [9] of a large corpus of database applications finds no evidence that applications manifestly change the default isolation level offered by the database. Taken together, these studies make clear that weakly-isolated transactions are quite prevalent in practice, whereas serializable transactions are often eschewed.

Unfortunately, weak isolation admits behaviours that are difficult to comprehend [12]. To quantify weak isolation anomalies, Fekete *et al.* [26] devised and experimented with a microbenchmark suite that executes transactions under a weakly-isolated *read committed* isolation level - the default level for 8 of the 18 databases studied in [6], and found that 25 out of every 1000 rows in the database violate at least one integrity constraint. Bailis *et al.* [9] rely on Rails' *uniqueness validation* to maintain uniqueness of records while serving Linkbench's [4] insertion workload (6400 records distributed over 1000 keys; 64 concurrent clients), and report discovering more than 10 duplicate records. Rails relies on database transactions to validate uniqueness during insertions, which is sensible if transactions are serializable, but incorrect under the *read committed* isolation level used in the experiments. The same study has found that 13% of all invariants among 67 open source Ruby-on-Rails applications are at risk of being violated due to weak isolation. Indeed, incidents of safety violations due to executing applications in a weakly-isolated environment have been reported on web services in production [43, 48], including in safety-critical applications such as bitcoin exchanges [16, 39]. While enforcing serializability for all transactions would be sufficient to avoid these errors and anomalies, it would likely be an overly conservative strategy; indeed, 75% of the invariants studied in [9] were shown to be preserved under some form of weak isolation. When to use weak isolation, and in what form, is therefore a prominent question facing all database programmers.

A major problem with weak isolation is that its semantics in the context of user programs is not well-understood. The original proposal [31] defines multiple "degrees" of weak isolation in terms of implementation details such as the nature and duration of locks held in each case. The ANSI SQL 92 standard defines four levels of isolation (including serializability) in terms of various undesirable *phenomena* (*e.g.,dirty reads*) each is required to prevent. While this is an improvement, it requires programmers to be prescient about

the possible ways various undesirable phenomena might manifest in their applications, and in each case determine if the phenomenon can be allowed without violating application invariants. This is understandably hard, especially in the absence of any formal underpinning to define weak isolation semantics. Adya [1] presents the first formal definitions of some well-known isolation levels in the context of a sequentially consistent (SC) database. However, there has been little progress relating Adya's system model to a formal operational semantics or a proof system that can facilitate rigorous correctness arguments. Consequently, reasoning about weak isolation remains an error prone endeavour, with major database vendors [37, 38, 40] continuing to document their isolation levels primarily in terms of the undesirable phenomena a particular isolation level may induce, placing the burden on the programmer to determine application correctness.

Recent results on reasoning about application invariants in the presence of weak consistency [10, 17, 30, 35, 36] address broadly related concerns. Weak consistency is a phenomenon that manifests on replicated data stores, where atomic operations are concurrently executed against different replicas, resulting in an execution order inconsistent with any sequential order. In contrast, weak isolation is a property of concurrent transactions interfering with one another resulting in an execution order that is not serializable. Unlike weak consistency, weak isolation can manifest even in an unreplicated setting, as evident from the support for weakly-isolated transactions on conventional (unreplicated) databases as mentioned above. In the presence of replication, however, the interaction between weak isolation and weak consistency can be subtle and non-trivial. Understanding weak isolation in these varied contexts thus requires new insights and substantial generalization of existing techniques.

In this paper, we propose a program logic for weakly-isolated transactions that realizes this goal. In particular, we develop a set of syntax-directed compositional proof rules that allow developers to build correctness proofs for transactional programs in the presence of a weakly-isolated concurrency control mechanism. A key novelty of our approach is that it is parametric over the *isolation semantics* of transactions in the program, as well as the *consistency semantics* of the underlying store. In concrete terms, this means that, unlike recent work focused on reasoning about programs under weak consistency [10, 30, 36], our system model does not assume a minimum or predefined set of consistency or isolation levels. Instead, our operational semantics admits declarative specifications of transaction isolation and store consistency, and generates executions that are guaranteed to satisfy these specifications. The result is a flexible system model that is general enough to incorporate the semantics of a range of isolation levels on a variety of stores (*e.g.,* the sequentially consistent store of [1], or the causally consistent store of [30], etc). Our key technical contribution is thus a set of proof rules that demonstrate that a program with a given selection of isolation levels for its transactions preserves its invariants when executed on a store equipped with appropriate consistency guarantees. The paper makes the following contributions:

1. We develop a semantics for a core language equipped with weakly-isolated transactions, demonstrating that a general parametric isolation semantics can be expressed as well-formedness constraints on a program's execution.
2. We present a compositional proof system for this language capable of relating high-level application invariants to the structure of traces induced by the operational semantics, in which transactions are associated with specific weak isolation levels.
3. We define a *maximum visibility principle* to reconcile store consistency guarantees with transaction isolation requirements, and use it to instantiate our operational model and the proof system for various kinds of (weakly-consistent) data stores.

```
                {B = k ∧ k ≥ a1+a2}
txn⟨Wd1⟩{                        ||   txn⟨Wd2⟩{
  if (B ≥ a1) {                  ||     if (B ≥ a2) {
    B := B - a1                  ||       B := B - a2
  }                              ||     }
}                                ||   }
                {B = k-a1-a2}
```

Figure 1: Concurrent withdraw transactions

4. Case studies modeled after real-world scenarios demonstrate the applicability and utility of our proof methodology.

Our results provide the first (to the best of our knowledge) mechanism that precisely and uniformly relates high-level program invariants to low-level weak isolation and weak consistency guarantees, thereby allowing weakly-isolated transactions to enjoy the same rigorous reasoning capabilities as their strongly-isolated (serializable) counterparts.

The remainder of the paper is organized as follows. The next section provides motivation and background on serializable and weakly-isolated transactions. §3 presents an operational semantics for a core language that supports weakly-isolated transactions, parameterized over different isolation notions. §4 formalizes the proof system that we use to reason about program invariants, and establishes the soundness of these rules with respect to the semantics. §5 generalizes the framework to integrate support for weakly-consistent data stores. We describe the impact of our reasoning framework in the context of several real-world case studies in §6. Related work and conclusions are given in §7.

2. Motivation

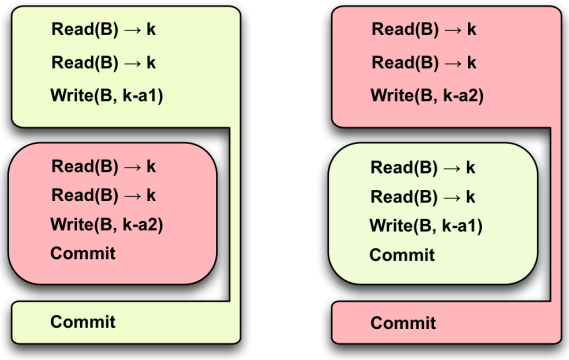
In this section, we motivate our ideas via an example written in $\mathcal{T}$- a C-like imperative language equipped with a `txn` lexical block defining a transaction scope. Each `txn` block is associated with an identifier in angle braces that uniquely identifies the transaction. We use Hoare triple notation to annotate programs with pre- and post- conditions.

Consider an implementation of a banking application that admits concurrent withdraw transactions on an account balance (`B`), as shown in Fig. 1. If the initial balance (`k`) in the account is enough to perform both withdraws, then the final balance, after both transactions commit, is expected to reflect the effects of both withdraws. The pre- and post-conditions in Fig. 1 reflect this expectation. Indeed, invariants are guaranteed to hold if both withdraw transactions are serialized, making *Serializable* isolation (SER) level a sufficient condition to preserve invariants. But, is SER necessary?

As an alternative, consider the execution of this transaction under a *Read Committed* (RC) isolation level, which is weaker than SER.[1] An RC transaction is isolated from the writes of uncommitted transactions, and is therefore free from *dirty reads* [12] of uncommitted data. In the current example, RC isolation admits the two executions shown in Fig. 2 on a strongly consistent (SC) store, such as a conventional RDBMS.

The figure depicts an execution as a series of read, write, and commit operations. In the execution on left, transaction `Wd1` (green) reads the current balance (`k`) and writes the new balance (`k-a1`), but before it commits, transaction `Wd2` (red) executes and commits, writing the new balance (`k-a2`). RC isolation prevent `Wd2` from witnessing the uncommitted writes of transaction `Wd1`. Subsequently committing `Wd1` leads to the loss of `Wd2`'s updates (the so-called *lost update anomaly* [12]), resulting in an incorrect balance of `k-a1`. The execution on right describes a similar scenario

[1] RC is in fact the default isolation level in Postgres 9.5 and Oracle 11g databases.



(a) RC Execution 1 (b) RC Execution 2

Figure 2: A possible execution of the program shown in Fig. 1 under *Read Committed* isolation level. Transaction `Wd1` is shown against lighter green background, and transaction `Wd2` against darker red background. Each transaction reads the balance (`B`) twice, hence two `Reads`.

with `Wd1` and `Wd2` exchanging their roles. Clearly, RC is an excessively weak isolation level for this program because it loses the updates of one transaction, resulting in the violation of the post-condition. A stronger isolation level that prevents lost updates is required. *Snapshot Isolation* (SI) [12] fits this requirement; SI effectively serializes transactions that update a shared data object by aborting and re-executing a transaction if write-write conflicts are detected during its commit.[2] Since SI, unlike SER, does not need expensive mechanisms such as lock-based concurrency control or runtime monitoring, it is also more efficient, making it appropriate for both withdraw transactions.

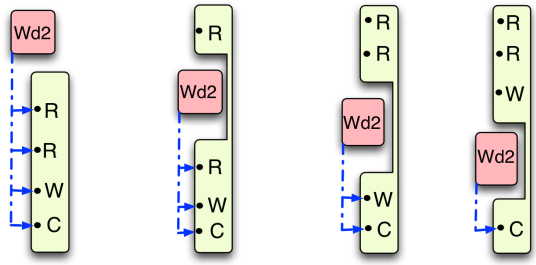
Thinking in terms of anomalies, as described above, is how database programmers are often encouraged to reason about weak isolation. Unfortunately, such reasoning does not rest on any sound foundation, and is thus highly error-prone. Reasoning in terms of how weak isolation variants are implemented is no better since it requires programmers to understand low-level implementation details of the database that are far removed from the application semantics. An attractive alternative in this context would be a principled reasoning approach that combines declarative reasoning about isolation guarantees with operational reasoning about programs. We demonstrate how our proof system makes this possible in the context of the current example.

First, we note that the example in Fig. 1 is a concurrent program, and hence is amenable to *rely-guarantee* style reasoning [33], a compositional proof technique that allows us to reason about the behaviour of individual threads by abstracting away interferences induced by other threads (collectively called *the environment*) into a *rely* relation. In an ordinary concurrent program, every environment step is a valid interference in the current thread. However, in the presence of transactions executing under various levels of weak isolation, determining what constitutes an interference is a non-trivial problem. For example, a *Serializable* transaction admits no interference, whereas within a transaction executing under *Read Uncommitted* isolation, all interferences are valid. Between these two extremes are various levels of isolation that admit some interferences while prohibiting others. For example, *Read Committed* isolation admits interference of a committed transaction, but not those from uncommitted transactions. *Snapshot Isolation* admits

$$\psi_{RC} \stackrel{def}{=} \forall T_1, T_2, \eta_1, \eta_2.\ \mathtt{txn}(\eta_1) = T_1 \wedge \mathtt{txn}(\eta_2) = T_2 \wedge T_1 \neq T_2 \wedge \eta_1 \xrightarrow{\mathsf{hb}} \eta_2 \Rightarrow T_1 \xrightarrow{\mathsf{hb}} \eta_2$$

$$\psi_{SI} \stackrel{def}{=} \forall T_1, T_2.\ T_1 \neq T_2 \wedge (\exists \mathtt{X}.\ T_1 \xrightarrow{\mathsf{wrsto}} \mathtt{X} \wedge T_2 \xrightarrow{\mathsf{wrsto}} \mathtt{X}) \Rightarrow T_1 \xrightarrow{\mathsf{hb}} T_2 \vee T_2 \xrightarrow{\mathsf{hb}} T_1$$

Figure 3: Interference properties for different weak isolation levels can be captured as constraints over the happens-before relation.



(a) (b) (c) (d)

Figure 4: Possible executions for the example in Fig. 1. Letters R, W and C stand for read, write and commit, respectively. Each blue dashed arrow represents a `hb` relationship between all operations of `Wd2` and an operation in `Wd1`. ψ_{SI} specification allows `hb` arrows (hence the execution) in Fig. 4a. ψ_{RC} specification allows `hb` arrows in Figs. 4a, 4b, 4c, and 4d.

interference from committed non-conflicting transactions, and so on.

For the rely-guarantee approach to be useful, it has to thus constrain allowed interference in accordance with the chosen level of isolation associated with a transaction. The first observation we make is that we can constrain interference by constraining the nature of the *happens-before* (`hb`) relation, which ultimately dictates what other transactions become visible to a transaction, or its constituents, and when. To do so, we axiomatize the `hb` relation to capture the interference characteristics of different isolation levels as their specifications. Two such specifications are shown[3] in Fig. 3. The *Read Committed* specification (ψ_{RC}) allows an operation η_1 of a transaction T_1 to happen-before an operation η_2 of another transaction T_2 only if every operation in T_1 (including its commit) happens-before η_2 (we let $T_1 \xrightarrow{\mathsf{hb}} \eta_2$ denote this). The specification does not require T_1 to execute and commit before all operations of T_2, thus allowing it to interfere in T_2, just as `Wd2` interferes in `Wd1` in Fig. 2a. The *Snapshot Isolation* specification (ψ_{SI}) requires two transactions that write to the same variable (denoted $\xrightarrow{\mathsf{wrsto}}$) to be related by happens-before. This effectively prohibits interference due to actions in T_1 being interleaved in T_2 (or, vice versa).

Fig. 4 is a visualization of `hb` from `Wd2` to `Wd1` allowed by ψ_{RC} and ψ_{SI}. Arrows in all executions are legal under ψ_{RC} because in no execution does an operation from `Wd2` happen-before an operation of `Wd1` without the commit of `Wd2` also happening before the operation of `Wd1`. ψ_{SI} however disallows `hb` arrows (hence executions) in Figs. 4b, 4c, and 4d because such arrows establish `hb` edges between (operations in) `Wd2` and only a subset of the operations in `Wd1`. Since the result of an operation (*e.g.,*a read) depends on what happens before that operation, the structure of the `hb` relation ultimately dictates the final result of the program. The arrows in Figs. 4a and 4b denote `hb` relationships that do not affect the value of `B` in a way that causes the program to violate

[2] SI however does not serialize transactions in the absence of write-write conflicts.

[3] Note that specifications presented here are solely for illustration. The actual specifications described in §3.1 are more nuanced.

its post-condition. In contrast, arrows in Figs. 4c and 4d denote hb relationships that lead to the violation of the post-condition. The task of the reasoning framework is to determine if *all* hb relationships allowed by an isolation level lead to the satisfaction of the post-condition.

The specification of an isolation level encodes its interference characteristics as constraints over the hb relation. However, for this to be useful in reasoning about programs, the rely-guarantee framework should be able to use the specification to determine if an interference is valid or not, allowing the programmer to only focus on valid interferences. Our second observation is that this is possible if the reasoning framework adequately tracks hb at each program point, while preserving hb constraints as invariants between program points. An interference that leads to the violation of hb constraints (i.e., an invalid interference) is thus automatically prohibited. For instance, consider the program point after the write to B in Wd1. The expected invariant (ϕ) at that program point is shown below (com stands for "committed"):

$$\neg\mathsf{com}(\texttt{Wd1}) \ \wedge\ \texttt{Wd1} \xrightarrow{\mathsf{wrsto}} \texttt{B} \ \wedge\ (\neg\mathsf{com}(\texttt{Wd2}) \Rightarrow \texttt{B} = \texttt{k} - \texttt{a1}) \\ \wedge\ (\mathsf{com}(\texttt{Wd2}) \Rightarrow \texttt{B} = \texttt{k} - \texttt{a1} - \texttt{a2})$$

ϕ asserts that Wd1 is not yet committed, and that it wrote to B, and the value of B is either k-a1-a2 or k-a1 depending on whether or not Wd2 is committed. If ϕ remains invariant until Wd1 commits, then the post-condition (B = k-a1-a2) can be established easily. However, an interference from Wd2 at this stage (captured by the last dashed arrow in Fig. 4) may violate the invariant by writing $\texttt{k} - \texttt{a2}$ to B and committing Wd2, thus leading to $\mathsf{com}(\texttt{Wd2}) \Rightarrow \texttt{B} = \texttt{k} - \texttt{a2}$. Fortunately, *Snapshot Isolation* prevents this interference, and this can be shown by demonstrating that an interference from Wd2 starting from an execution state that satisfies $\psi_{SI} \wedge \phi$ leads to an execution state where neither $\texttt{Wd2} \xrightarrow{\mathsf{hb}} \texttt{Wd1}$ nor $\texttt{Wd1} \xrightarrow{\mathsf{hb}} \texttt{Wd2}$ holds; $\texttt{Wd2} \xrightarrow{\mathsf{hb}} \texttt{Wd1}$ does not hold because Wd1's write to B clearly happened before Wd2's commit, and $\texttt{Wd1} \xrightarrow{\mathsf{hb}} \texttt{Wd2}$ does not hold because Wd1 has not yet committed, while Wd2 has already begun. Since Wd2 also writes to B, this violates the ψ_{SI} constraint which we assume to be an invariant. A proof that the post-condition holds now follows from the contradiction. It is informative to note that if the invariant is ψ_{RC} instead of ψ_{SI}, we cannot derive a contradiction and we cannot rule out the interference, which (rightfully) causes the proof to fail.

Implicit in the above discussion is the assumption of a strongly consistent (SC) store that guarantees the visibility of all previously committed transactions. An SC semantics can be built into the reasoning framework, leading to a proof system tailor-made for such stores. However, for the reasoning framework to be truly useful, it should be capable of handling different consistency semantics, supporting stores that are weaker than SC (e.g., causally or eventually consistent), and should be able to reconcile conflicts between consistency and isolation constraints. We demonstrate how our reasoning framework makes this possible in the following sections.

3. $\mathcal{T}$: Syntax and Semantics

Fig. 5 shows the syntax and small-step semantics of $\mathcal{T}$, a core language that we will use to formalize the intuitions presented in the previous section. Natural numbers, (shared) variables and arithmetic expressions constitute the syntactic class of expressions (e). Commands (c) include SKIP, assignment statements, transaction (txn) lexical blocks, and their sequential and parallel composition. We let T_i for $i \in \mathbb{N}$ range over transaction identifiers. When it is evident we are referring to a transaction, we use the number i instead of T_i for identification (*e.g.,*in $\texttt{txn}\langle i \rangle$). Like variables, transaction identifiers are globally accessible. For notational convenience, we let t range over both expressions and commands.

We define a small-step operational semantics for this language in terms of an abstract machine that generates an execution trace (E). The first component of the trace is a set (A) of *effects*, where an effect (η) is a tuple of (a). the identifier (T_i) of the transaction that generated this effect, (b). the unique identifier (j) of this effect, (c). the operation (op) documented by this effect, which can be either a read (RD(X)), a write (WR(X)), or a transaction commit (COMMIT), and (d). the value, if any, associated with the operation (*e.g.,*, value read or value written). Functions txn, id, oper, and rval project each of the four aforementioned individual components of the tuple. In every step of the evaluation, the machine reduces a $\mathcal{T}$ term by executing a read, write or commit operation, generating an effect, and extending the trace. Since effects include transaction identifiers, the semantics distinguishes between terms (t) of different transactions. For example, $[t]_i$ denotes a term t inside a transaction T_i. Evaluation contexts are also appropriately marked. For example, $\mathcal{E}_i$ denotes the evaluation context for a term inside T_i. The other component of an execution trace is a visibility relation (vis) that establishes a visibility property between effects among different transactions. The intent and mechanics of vis is described in the sequel.

Fundamental to our development is the notion of a trace invariant ($\mathbb{I}$). $\mathbb{I}$ is a function from traces (E) to first-order logical formulas ($\mathbb{P}$) that define well-formedness constraints over traces. The machine takes a step only if the resulting trace satisfies the constraints imposed by $\mathbb{I}$. This behaviour is captured by the auxiliary reduction rule E-AUX that factors out the trace extension aspect of the evaluation by abstracting away the operation-specific behaviour as a function that generates an appropriate effect. We let $\mathcal{F}$ denote this function. E-AUX uses $\mathcal{F}$ to generate a new effect and extend the trace ($\mathsf{E} = (\mathsf{A}, \mathsf{vis})$) *only if* the well-formedness constraints imposed by $\mathbb{I}$ on E (i.e., $\mathbb{I}(\mathsf{E}')$) are satisfied. Otherwise, it gets stuck. In an execution that runs to completion, every small-step preserves the well-formedness of a trace, thus ensuring the invariance of $\mathbb{I}$. Note that the semantics makes no assumptions about $\mathbb{I}$ other than its type. As such, it can be instantiated with any trace-parametric proposition that expresses constraints over the given trace. For instance, consider the ψ_{RC} specification from §2, but with bounded T_1 and T_2 instantiated with Wd1 and Wd2, respectively. The instantiated specification is the following term:

$$\forall \eta_1, \eta_2.\ \mathsf{txn}(\eta_1) = \texttt{Wd1} \ \wedge\ \mathsf{txn}(\eta_2) = \texttt{Wd2} \\ \wedge\ \texttt{Wd1} \neq \texttt{Wd2} \ \wedge\ \eta_1 \xrightarrow{\mathsf{hb}} \eta_2 \Rightarrow \texttt{Wd1} \xrightarrow{\mathsf{hb}} \eta_2$$

It is easy to interpret the above specification in the context of a trace E that captures an execution of the program in Fig. 1. Such a trace-parametric formula can be used to instantiate the trace invariant $\mathbb{I}$ in Fig. 5. The resultant operational semantics describes an abstract machine that gets stuck if an operation of Wd2 is executed in a state that incorporates some, but not all the effects (including the COMMIT) of Wd1.

As described in §2, the semantics of various isolation levels can be captured as constraints over the happens-before (hb) relation. hb is however a derived relation in our model, composed of more fundamental *session order* (so) and *visibility* (vis) relations. In particular, $\mathsf{hb} = (\mathsf{so} \cup \mathsf{vis})^+$. Unlike hb, vis (defined below) is not a transitive relation, and hence lets us capture finer-grained isolation properties than hb, which we leverage in our development. The session order relation captures the sequential order of operations within a transaction. In particular, it relates two effects, η_1 and η_2, such that $\mathsf{txn}(\eta_1) = \mathsf{txn}(\eta_2)$ and $\mathsf{id}(\eta_1) < \mathsf{id}(\eta_2)$. The semantics assigns monotonically increasing identifiers to effects, as defined by the $\mathsf{id}(\eta) > \mathsf{maxId}(\mathsf{A})$ condition of E-AUX ($\mathsf{maxId}(\mathsf{A})$ returns the maximum number identifying an effect in A). Evaluation contexts ($\mathcal{E}_i$) for transaction-bound terms are defined so as to enforce a deterministic sequential order of execution within a transaction, leading to a deterministic total order among effect ids, which de-

Syntax

$$X, Y \in \texttt{Id} \qquad i, j, n, v \in \mathbb{N} \qquad T_i, T_j \in \texttt{TxnId}$$

$$\begin{array}{lcll} e & \in & \texttt{Expressions} & ::= \quad v \mid X \mid e \pm e \\ c & \in & \texttt{Commands} & ::= \quad \texttt{SKIP} \mid X := e \mid c_1; c_2 \mid c1 || c2 \mid \texttt{txn}\langle i \rangle \{c\} \\ t & \in & \texttt{Terms} & ::= \quad e \mid c \end{array}$$

Auxiliary Definitions

$$\begin{array}{llll} \mathsf{op} \in \texttt{Op. Name} & ::= & \texttt{RD}(X) \mid \texttt{WR}(X) \mid \texttt{COMMIT} & \\ \eta \in \texttt{Effect} & ::= & (T_i, j, \mathsf{op}, \mathbb{N} \cup \{\bot\}) & \\ \mathsf{A} \in \texttt{Effect Set} & ::= & \mathcal{P}(\texttt{Effect}) & \\ [\![\cdot]\!] \in \texttt{State} & ::= & \mathcal{P}(\texttt{Effect}) \to \texttt{Id} \to \mathbb{N} & \end{array} \qquad \begin{array}{llll} \mathsf{vis} \in \texttt{Visibility} & ::= & \mathcal{P}(\texttt{Effect} \times \texttt{Effect}) & \\ \mathsf{E} \in \texttt{Execution} & ::= & (\mathsf{A}, \mathsf{vis}) & \\ \mathbb{I} \in \texttt{Invariant} & ::= & \mathsf{E} \to \mathbb{P} & \\ \mathcal{F} \in \texttt{Op. Def.} & ::= & \mathcal{P}(\texttt{Effect}) \to \texttt{Effect} & \end{array}$$

$$\mathsf{txn}(t_i, \_, \_, \_) = t_i \qquad \mathsf{id}(\_, j, \_, \_) = j \qquad \mathsf{oper}(\_, \_, \mathsf{op}, \_) = \mathsf{op} \qquad \mathsf{rval}(\_, \_, \_, n) = n$$

Evaluation $\boxed{\mathbb{I} \vdash (\mathcal{F}, \mathsf{E}) \hookrightarrow (v, \mathsf{E}')}$ $\boxed{\mathbb{I} \vdash ([t]_i, \mathsf{E}) \longrightarrow (t', \mathsf{E}')}$ $\boxed{\mathbb{I} \vdash (c, \mathsf{E}) \longrightarrow (c', \mathsf{E}')}$

$$\mathcal{E} \in \texttt{Eval Ctx} ::= \bullet \mid \bullet \,||\, c_2 \mid c_1 \,||\, \bullet \qquad \mathcal{E}_i \in \texttt{Eval Ctx in txn}\langle i \rangle ::= \texttt{txn}\langle i \rangle \{\bullet\} \mid [\bullet \pm e_2]_i \mid [v_1 \pm \bullet]_i \mid [\bullet; c_2]_i$$

E-AUX

$$\frac{S \subseteq \mathsf{A} \quad \eta = \mathcal{F}(S) \quad \mathsf{id}(\eta) > \mathsf{maxId}(\mathsf{A}) \quad \mathsf{E}' = (\mathsf{A} \cup \{\eta\}, \mathsf{vis} \cup S \times \{\eta\}) \quad \mathbb{I}(\mathsf{E}')}{\mathbb{I} \vdash (\mathcal{F}, (\mathsf{A}, \mathsf{vis})) \hookrightarrow (\mathsf{rval}(\eta), \mathsf{E}')}$$

E-TOP-CTX

$$\frac{\mathbb{I} \vdash (t, \mathsf{E}) \longrightarrow (t', \mathsf{E}')}{\mathbb{I} \vdash (\mathcal{E}[t], \mathsf{E}) \longrightarrow (\mathcal{E}[t'], \mathsf{E}')}$$

E-TXN-CTX

$$\frac{\mathbb{I} \vdash ([t]_i, \mathsf{E}) \longrightarrow (t', \mathsf{E}')}{\mathbb{I} \vdash (\mathcal{E}_i[t], \mathsf{E}) \longrightarrow (\mathcal{E}_i[t'], \mathsf{E}')}$$

E-VAR

$$\frac{\mathcal{F} = \lambda S.\, (T_i, j, \texttt{RD}(X), [\![S]\!](X)) \quad \mathbb{I} \vdash (\mathcal{F}, \mathsf{E}) \hookrightarrow (v, \mathsf{E}')}{\mathbb{I} \vdash ([X]_i, \mathsf{E}) \longrightarrow (v, \mathsf{E}')}$$

E-ASGN

$$\frac{\mathcal{F} = \lambda S.\, (T_i, j, \texttt{WR(X)}, v) \quad \mathbb{I} \vdash (\mathcal{F}, \mathsf{E}) \hookrightarrow (v, \mathsf{E}')}{\mathbb{I} \vdash ([X := v]_i, \mathsf{E}) \longrightarrow (\texttt{SKIP}, \mathsf{E}')}$$

E-COMMIT

$$\frac{\mathcal{F} = \lambda S.\, (T_i, j, \texttt{COMMIT}, \bot) \quad \mathbb{I} \vdash (\mathcal{F}, \mathsf{E}) \hookrightarrow (\bot, \mathsf{E}')}{\mathbb{I} \vdash (\texttt{txn}\langle i \rangle \{\texttt{SKIP}\}, \mathsf{E}) \longrightarrow (\texttt{SKIP}, \mathsf{E}')}$$

Figure 5: $\mathcal{T}$: Syntax and Small-step semantics

fines the session order relation. Visibility (vis) on the other hand relates effects across concurrent transactions. Intuitively, vis relates η_1 to η_2 if and only if η_1 was *visible* to the operation that generated η_2 during its execution, thus effecting its return value ($\mathsf{rval}(\eta_2)$). For example, a read operation over `X` may pick the value (rval) of the write effect with the highest id among its visible effects (this is made possible by appropriately defining $[\![\cdot]\!]$ in E-VAR, as we show later). Thus, the value of a read depends on what write effects it can witness. An operation can only witness the effects of already concluded operations, which varies between executions due to the non-deterministic order of evaluating the parallel composition of transactions.

A more notable source of non-determinism, however, is the E-AUX rule, which allows the machine to expose an arbitrary subset (S) of existing effects (A) to the incoming operation. In other words, the machine is not obligated to reveal the effects of all previous operations to an incoming operation. This relaxation allows the abstract machine to model the semantics of weakly-consistent data stores. For instance, operations issued to an eventually consistent (EC) replicated store could be dispatched to different replicas whose states may not be in any well-defined relationship. By allowing operations to witness arbitrary subsets of the global state, the semantics models the weak visibility properties of such stores; we elaborate on the implication of this style of definition in §5. Stronger visibility properties can be expressed by imposing well-formedness constraints over vis via the trace invariant ($\mathbb{I}$). Since the abstract machine is obligated to satisfy $\mathbb{I}$ at every step of the execution, operations are guaranteed to experience the level of isolation specified by $\mathbb{I}$. Thus, in executions that run to completion, the abstract machine models a store that provides the required levels of isolation. Notably, the machine achieves this without defining an operational semantics for isolation levels, instead solely relying on their declarative characterization as trace well-formedness constraints to enforce isolation guarantees. §3.1 specifies various ANSI SQL isolation levels stated as trace well-formedness constraints.

As described previously, E-AUX abstracts away the operation-specific behaviour of a machine step as a function ($\mathcal{F}$) that accepts a set (S) of effects chosen by the machine to make visible to the operation, interprets the operation w.r.t. S, and returns an appropriate effect that encodes its return value. Rules E-VAR, E-ASGN and E-COMMIT define such functions for read, write and commit operations, respectively. The effect returned by the function in each case includes its transaction id (T_i) along with an arbitrarily chosen effect id (j) that is later verified to be unique in E-AUX. The rval for a write is the value being written, and for commit it is $\bot$. In case of a read, the value read depends on how the read operation chooses to interpret the given set (S) of visible effects. The interpretation may depend on application semantics. For example, a monotonically increasing counter application may choose to let a write with the largest value determine the value of a read. To accommodate multiple interpretations, the semantics is made parametric over an interpretation function ($[\![\cdot]\!]$) that accepts a set of effects and a variable name, and returns the value associated with the variable. A straightforward interpretation function that chooses the last write (i.e., write with largest id) is shown below:

$$\begin{array}{lcl} \mathsf{isMax}(S, \eta) & \Leftrightarrow & \forall (\eta' \in S).\mathsf{oper}(\eta') = \mathsf{oper}(\eta) \\ & & \qquad \Rightarrow \eta' = \eta \ \vee \ \mathsf{id}(\eta') < \mathsf{id}(\eta) \\ [\![S]\!](X) & = & \texttt{if } (\exists (\eta \in S).\mathsf{oper}(\eta) = \texttt{WR}(X) \wedge \mathsf{isMax}(S, \eta)) \\ & & \texttt{then } \mathsf{rval}(\eta) \texttt{ else } 0 \end{array}$$

Rules E-TOP-CTX and E-TXN-CTX define congruence properties for top-level terms and transaction-bound terms, respectively. The rules and evaluation contexts ($\mathcal{E}$ and $\mathcal{E}_i$) are defined such that only certain kinds of terms are allowed at the top-level and inside a

$$
\begin{array}{lcl}
(\mathsf{A},\mathsf{vis}) \Vdash \eta \in T_i & \stackrel{def}{=} & \eta \in \mathsf{A} \ \wedge \ \mathsf{txn}(\eta) = T_i \\
\mathsf{E} \Vdash S \subseteq T_i & \stackrel{def}{=} & \forall \eta.\ \eta \in S \Rightarrow \mathsf{E} \Vdash \eta \in T_i \\
(\mathsf{A},\mathsf{vis}) \Vdash \eta_1 \xrightarrow{\mathsf{vis}} \eta_2 & \stackrel{def}{=} & \{\eta_1,\eta_2\} \subseteq \mathsf{A} \ \wedge \ (\eta_1,\eta_2) \in \mathsf{vis} \\
(\mathsf{A},\mathsf{vis}) \Vdash \eta_1 \xrightarrow{\mathsf{so}} \eta_2 & \stackrel{def}{=} & \{\eta_1,\eta_2\} \subseteq \mathsf{A} \ \wedge \ \mathsf{txn}(\eta_1) = \mathsf{txn}(\eta_2) \\
 & & \qquad \wedge \ \mathsf{id}(\eta_1) < \mathsf{id}(\eta_2) \\
(\mathsf{A},\mathsf{vis}) \Vdash \eta_1 \xrightarrow{\mathsf{hb}} \eta_2 & \stackrel{def}{=} & \{\eta_1,\eta_2\} \subseteq \mathsf{A} \ \wedge \ (\eta_1,\eta_2) \in (\mathsf{vis} \cup \mathsf{so})^+ \\
\mathsf{E} \Vdash T_i \xrightarrow{\mathsf{vis}} \eta & \stackrel{def}{=} & \forall \eta_1.\ (\mathsf{E} \Vdash \eta_1 \in T_i) \Rightarrow \mathsf{E} \Vdash \eta_1 \xrightarrow{\mathsf{vis}} \eta
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{E} \Vdash T_i \xrightarrow{\mathsf{vis}} T_j & \stackrel{def}{=} & \forall \eta_1,\eta_2.\ \mathsf{E} \Vdash \eta_1 \in T_i \ \wedge \ \mathsf{E} \Vdash \eta_2 \in T_j \\
 & & \qquad \Rightarrow \mathsf{E} \Vdash \eta_1 \xrightarrow{\mathsf{vis}} \eta_2 \\
\mathsf{E} \Vdash T_i \xrightarrow{\mathsf{invis}} T_j & \stackrel{def}{=} & \forall \eta_1,\eta_2.\ \mathsf{E} \Vdash \eta_1 \in T_i \ \wedge \ \mathsf{E} \Vdash \eta_2 \in T_j \\
 & & \qquad \Rightarrow \neg(\mathsf{E} \Vdash \eta_1 \xrightarrow{\mathsf{vis}} \eta_2) \\
\mathsf{E} \Vdash T_i \xrightarrow{\mathsf{wrsto}} X & \stackrel{def}{=} & \exists \eta.\ \mathsf{E} \Vdash \eta \in T_i \ \wedge \ \mathsf{oper}(\eta) = \mathtt{WR}(X) \\
\mathsf{E} \Vdash T_i \xrightarrow{\mathsf{rdsfm}} X & \stackrel{def}{=} & \exists \eta.\ \mathsf{E} \Vdash \eta \in T_i \ \wedge \ \mathsf{oper}(\eta) = \mathtt{RD}(X) \\
\mathsf{E} \Vdash T_i \xrightarrow{\mathsf{uses}} X & \stackrel{def}{=} & \mathsf{E} \Vdash T_i \xrightarrow{\mathsf{wrsto}} X \ \vee \ \mathsf{E} \Vdash T_i \xrightarrow{\mathsf{rdsfm}} X
\end{array}
$$

Figure 6: Relations defined over a trace

transaction. In particular, a $\mathcal{T}$program at the top-level can either be a transaction, or a parallel composition of transactions. A command inside a `txn` block can either be an assignment, or a sequential composition of assignments.

3.1 Isolation Specifications

We now describe specifications of standard isolation guarantees expressed as constraints over trace well-formedness. For brevity and convenience, we introduce some notations that are used in the following sections. An execution trace is destructed as (A,vis) whenever individual components of the pair are needed. Otherwise, it is written as E. Sometimes, the dot notation (*e.g.,* E.A) is also used. Since A and vis are both sets, we lift the operations on sets to pairs of sets when updating E. For example, $\mathsf{E}' = \mathsf{E} \cup (\{\eta_2\}, \{(\eta_1,\eta_2)\})$ expands to $\mathsf{E}' = (\mathsf{E}.\mathsf{A} \cup \{\eta_2\},\ \mathsf{E}.\mathsf{vis} \cup \{(\eta_1,\eta_2)\})$. When ψ is a formula, $\mathsf{E} \Vdash \psi$ denotes the interpretation of ψ in the context of the trace E. Such interpretations are defined on a case-by-case basis in Figs. 6 and 7.

Fig. 6 shows various relations defined over elements in a trace. In the context of a trace $(\mathsf{A},\mathsf{vis})$, an effect η is said to belong to a transaction T_i if η belongs to the effect set A and its transaction identifier is T_i. The containment relation is trivially lifted to the set of effects to define $\mathsf{E} \Vdash S \subseteq T_i$. Visibility, session order, and happens-before relations are denoted by $\eta_1 \xrightarrow{\mathsf{vis}} \eta_2$, $\eta_1 \xrightarrow{\mathsf{so}} \eta_2$, and $\eta_1 \xrightarrow{\mathsf{hb}} \eta_2$, respectively. A transaction T_i is said to be visible to an effect η if every effect η_1 of T_i recorded by the trace is visible to η. T_i may be visible to η but may not be visible to every other effect in the $\mathsf{txn}(\eta)$. For a transaction T_i to be considered to be visible to a transaction T_j in the context of a trace E (written $\mathsf{E} \Vdash T_i \xrightarrow{\mathsf{vis}} T_j$), every effect ($\eta_1$) of T_i present in E must be visible to every effect (η_2) of T_j in E. Conversely, if none of the effects of T_i present in E are visible to any effect of T_j, then T_i is considered invisible to T_j under E (written $\mathsf{E} \Vdash T_i \xrightarrow{\mathsf{invis}} T_j$). Transaction T_i is said to have written to a variable X under E (i.e., $\mathsf{E} \Vdash T_i \xrightarrow{\mathsf{wrsto}} X$) if there exists a $\mathtt{WR}(X)$ effect from T_i in E. *Reads-from* ($\mathsf{E} \Vdash T_i \xrightarrow{\mathsf{rdsfm}} X$) is defined similarly. T_i *uses* X ($T_i \xrightarrow{\mathsf{uses}} X$) if it reads or writes X.

Fig. 7 shows various isolation guarantees defined as propositions indexed by transaction identifiers. Transaction T_j is said to have experienced *read-my-writes* visibility (`RMWVis`) under E if every effect (η_1) of T_j is visible to every subsequent effect (η_2) of the same transaction in E. This lets T_j to never lose its own updates. Monotonic visibility (`MonotonicVis`) adds one more constraint to *read-my-writes*; besides requiring η_1 to be visible to η_2, it also requires every effect (η) visible to η_1 in E to be visible to η_2 as well. Thus, later operations in a transaction witness at least the same set of effects witnessed by the earlier operations, if not more (hence, "monotonic" visibility). Atomic visibility (`AtomicVis`) allows an effect η_2 of T_j to witness an effect η_1 of T_i only if all effects of T_i in E are also visible to η_2. Atomic visibility thus prevents a transaction from being partially visible. However, atomic visibility does not prevent an uncommitted transaction from being visible. This is addressed by `CommitVis`, which requires the commit effect of T_i to be visible whenever any effect of T_i is visible.

The ANSI SQL 92 standard requires *Read Committed* isolation to avoid the dirty reads phenomenon, which is achieved by enforcing `AtomicVis` and `CommitVis` guarantees. The RC specification (`RC`)[4] is therefore a combination of these two guarantees. The specification also agrees with the description and implementation [5, 46] of RC for highly available replicated stores. On relational databases, however, RC has also come to be associated with the `MonotonicVis` guarantee. Nonetheless, `AtomicVis` and `CommitVis` are sufficient to reason about RC isolation on relational stores too. The combination of these guarantees with the SC property of relational stores (formalized in §5) automatically leads to the monotonicity guarantee, which explains why RC comes with `MonotonicVis` on such stores regardless of the implementation. On weakly consistent stores however, `AtomicVis` and `CommitVis` do not imply `MonotonicVis`. A stronger isolation level called *Monotonic Atomic View* (MAV of Fig. 7) [5] was proposed to explicitly extend RC with monotonicity on such stores.

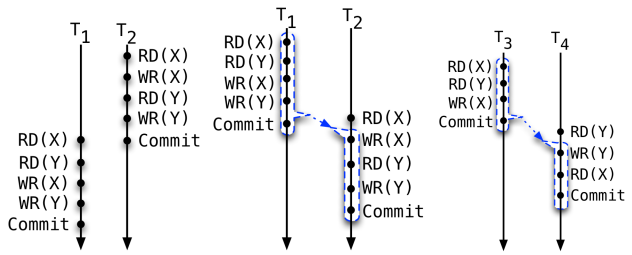
Snapshot visibility (`SnapshotVis`) captures the scenario where a transaction executes against a static snapshot of the database. A transaction T_i is said to be snapshot-visible to a transaction T_j if either it is visible to T_j (i.e., T_i is included in the snapshot), or it is invisible (i.e., it is not included); it is forbidden for only a suffix (more generally, a subset) of T_j to witness T_i. The specification of ANSI SQL *Repeatable Read* isolation (`RR`) extends `MAV` with the snapshot visibility guarantee. Observe that snapshot visibility permits T_i and T_j to execute and commit while being oblivious of each other. This scenario is captured in Fig. 8a, where transactions T_1 and T_2, which perform conflicting writes, execute against a snapshot of the database and commit concurrently. While the actual values read and written by T_1 and T_2 are unimportant (hence, elided), it is important to note the *absence* of visibility arrows from T_2 to T_1, although T_2 commits before T_1's read-from-Y.

Snapshot Isolation (SI) proscribes this possibility. If T_i and T_j both write to the same shared variable, then SI insists that either T_i be visible to T_j, or T_j be visible to the conflicting write of T_i (this is captured by the auxiliary definition `SnapshotSER`). Fig. 8b shows an execution where SI transaction T_1 witnesses a snapshot of the database that doesn't include T_2. Due to `SI`(T_1), the conflicting write to X in T_2 is now required to witness T_1, as captured by the direction of the vis arrow (subsequent operations also witness T_1 because T_2 is executing under MAV). Lastly, *Serializable* isolation extends `SnapshotSER` to also cover non-conflicting transactions that write to variables read in the current transaction. Fig. 8c shows an execution of an SER transaction T_3 and a MAV transaction T_4,

[4] In the following, we use small caps to abbreviate isolation levels (*e.g.,* RC for *Read Committed*), and typewriter font for abbreviations of the specification of an isolation level (*e.g.,* `RC` denotes the specification for RC as given in Fig. 7).

$$
\begin{array}{lcl}
\mathsf{E} \Vdash \texttt{RMWVis}(T_j) & \stackrel{def}{=} & \forall \eta_1, \eta_2.\, \mathsf{E} \Vdash \{\eta_1, \eta_2\} \subseteq T_j \ \wedge\ \mathsf{E} \Vdash \eta_1 \xrightarrow{\mathsf{so}} \eta_2 \Rightarrow \mathsf{E} \Vdash \eta_1 \xrightarrow{\mathsf{vis}} \eta_2 \\
\mathsf{E} \Vdash \texttt{MonotonicVis}(T_j) & \stackrel{def}{=} & \mathsf{E} \Vdash \texttt{RMWVis}(T_j) \ \wedge\ \forall \eta, \eta_1, \eta_2.\, \mathsf{E} \Vdash \{\eta_1, \eta_2\} \in T_j \ \wedge \\
& & \qquad \mathsf{E} \Vdash \eta \xrightarrow{\mathsf{vis}} \eta_1 \ \wedge\ \mathsf{E} \Vdash \eta_1 \xrightarrow{\mathsf{so}} \eta_2 \Rightarrow \mathsf{E} \Vdash \eta \xrightarrow{\mathsf{vis}} \eta_2 \\
\mathsf{E} \Vdash \texttt{AtomicVis}(T_j) & \stackrel{def}{=} & \forall \eta_1, \eta_2.\, \neg(\mathsf{E} \Vdash \eta_1 \in T_j) \ \wedge\ \mathsf{E} \Vdash \eta_2 \in T_j \ \wedge\ \mathsf{E} \Vdash \eta_1 \xrightarrow{\mathsf{vis}} \eta_2 \Rightarrow \mathsf{E} \Vdash \mathsf{txn}(\eta_1) \xrightarrow{\mathsf{vis}} \eta_2 \\
\mathsf{E} \Vdash \texttt{CommitVis}(T_j) & \stackrel{def}{=} & \forall \eta_1, \eta_2.\, \neg(\mathsf{E} \Vdash \eta_1 \in T_j) \ \wedge\ \mathsf{E} \Vdash \eta_2 \in T_j \ \wedge\ \mathsf{E} \Vdash \eta_1 \xrightarrow{\mathsf{vis}} \eta_2 \Rightarrow \\
& & \qquad \exists \eta.\, \mathsf{E} \Vdash \eta \in \mathsf{txn}(\eta_1) \ \wedge\ \mathsf{oper}(\eta) = \texttt{COMMIT} \ \wedge\ \mathsf{E} \Vdash \eta \xrightarrow{\mathsf{vis}} \eta_2 \\
\mathsf{E} \Vdash \texttt{RC}(T_j) & \stackrel{def}{=} & \mathsf{E} \Vdash \texttt{AtomicVis}(T_j) \ \wedge\ \mathsf{E} \Vdash \texttt{CommitVis}(T_j) \\
\mathsf{E} \Vdash \texttt{MAV}(T_j) & \stackrel{def}{=} & \mathsf{E} \Vdash \texttt{RC}(T_j) \ \wedge\ \mathsf{E} \Vdash \texttt{MonotonicVis}(T_j) \\
\mathsf{E} \Vdash \texttt{SnapshotVis}(T_i, T_j) & \stackrel{def}{=} & \mathsf{E} \Vdash T_i \xrightarrow{\mathsf{vis}} T_j \ \vee\ \mathsf{E} \Vdash T_i \xrightarrow{\mathsf{invis}} T_j \\
\mathsf{E} \Vdash \texttt{RR}(T_j) & \stackrel{def}{=} & \mathsf{E} \Vdash \texttt{MAV}(T_j) \ \wedge\ \forall T_i.\, T_i \neq T_j \Rightarrow \mathsf{E} \Vdash \texttt{SnapshotVis}(T_i, T_j) \\
\mathsf{E} \Vdash \texttt{SnapshotSER}(T_i, T_j, X) & \stackrel{def}{=} & \mathsf{E} \Vdash T_i \xrightarrow{\mathsf{vis}} T_j \ \vee\ (\mathsf{E} \Vdash T_i \xrightarrow{\mathsf{invis}} T_j \ \wedge \\
& & \qquad \exists \eta.\, \mathsf{E} \Vdash \eta \in T_i \ \wedge\ \mathsf{oper}(\eta) = \texttt{WR(X)} \Rightarrow \mathsf{E} \Vdash T_j \xrightarrow{\mathsf{vis}} \eta) \\
\mathsf{E} \Vdash \texttt{SI}(T_j) & \stackrel{def}{=} & \mathsf{E} \Vdash \texttt{RR}(T_j) \ \wedge\ \forall T_i.\, (T_i \neq T_j \ \wedge\ \exists X.\, \mathsf{E} \Vdash T_i \xrightarrow{\mathsf{wrsto}} X \ \wedge\ \mathsf{E} \Vdash T_j \xrightarrow{\mathsf{wrsto}} X) \\
& & \qquad \Rightarrow \mathsf{E} \Vdash \texttt{SnapshotSER}(T_i, T_j, X) \\
\mathsf{E} \Vdash \texttt{SER}(T_j) & \stackrel{def}{=} & \mathsf{E} \Vdash \texttt{RR}(T_j) \ \wedge\ \forall T_i.\, (T_i \neq T_j \ \wedge\ \exists X.\, \mathsf{E} \Vdash T_i \xrightarrow{\mathsf{wrsto}} X \ \wedge\ \mathsf{E} \Vdash T_j \xrightarrow{\mathsf{uses}} X) \\
& & \qquad \Rightarrow \mathsf{E} \Vdash \texttt{SnapshotSER}(T_i, T_j, X)
\end{array}
$$

Figure 7: Standard isolation guarantees expressed as trace well-formedness constraints



(a) RR(T_1), RR(T_2). (b) SI(T_1), MAV(T_2) (c) SER(T_3), MAV(T_4)

Figure 8: Sample executions of concurrent transactions at different isolation levels. Solid (black) arrows indicate the timeline. Points on the timeline mark the time when an operation is executed. Dotted (blue) arrows denote vis.

which write to X and Y, respectively. The snapshot witnessed by the SER transaction T_3 does not include T_4, but the write to Y in T_4, although non-conflicting, witnesses T_1 because Y is read by T_1. Visibility includes other operations of T_4 because of MAV. Note that SER does not guarantee a total order on transactions w.r.t. vis because, in practice, databases do not guarantee global serializable execution unless *all* transactions choose SER.

4. The Reasoning Framework

We now describe a proof system that lets us demonstrate that a $\mathcal{T}$ program satisfies its high-level invariants in all small-step executions that satisfy a chosen trace invariant ($\mathbb{I}$).[5] Since the trace invariant captures transaction-isolation constraints, the demonstration is a proof that the given selection of isolation levels for transactions is sufficient to enforce a program's invariants.

As mentioned earlier, our proof system is a rely-guarantee framework that admits compositional reasoning by abstracting away environmental interference as a rely relation. A conspicuous difference between a standard development of rely-guarantee and ours is that, while the former reasons in terms of program states (variable to value bindings), we reason in terms of executions as captured by their traces (E). In particular, our rely (R) and guarantee (G) relations relate executions (i.e., $R,\ G\ \subseteq\ \mathsf{E} \times \mathsf{E}$), and our pre- ($P$) and post- ($Q$) conditions are assertions over executions (i.e., $P, Q\ :\ \mathsf{E}\ \rightarrow\ \mathbb{P}$). Our development also facilitates state-based reasoning via the interpretation function ($[\![\cdot]\!]$) introduced in §3, which interprets the given set of effects as a state. For example, if a *bi-state* rely relation relates every pair of states σ and σ' such that $\sigma'(X) \geq \sigma(X) \geq 0$, the corresponding *bi-execution* rely relation relates every pair of executions E and E' such that $[\![\mathsf{E}.\mathsf{A}]\!](X) \geq [\![\mathsf{E}'.\mathsf{A}]\!](X) \geq 0$. Assertions on states are also written similarly. For instance, consider the post-condition $\mathtt{B} = \mathtt{k} - \mathtt{a1} - \mathtt{a2}$ of the program in Fig. 1. The corresponding assertion on the post-state ($\lambda\sigma.\ \sigma(\mathtt{B}) = \mathtt{k} - \mathtt{a1} - \mathtt{a2}$) asserts that in all states resulting from executing the program, the value of B is k-a1-a2. The equivalent execution-based assertion ($\lambda\mathsf{E}.\ [\![\mathsf{E}.\mathsf{A}]\!](\mathtt{B}) = \mathtt{k} - \mathtt{a1} - \mathtt{a2}$) asserts that in all executions of the program, the value of B is k-a1-a2. However, having access to an execution facilitates assertions that go beyond the state. An example is an invariant of Wd1 described abstractly in §2, and reified as an execution-based assertion below:

$$
\begin{array}{l}
\lambda\mathsf{E}.\ \neg(\mathsf{E} \Vdash \mathsf{com}(\texttt{Wd1}) \ \wedge\ \mathsf{E} \Vdash \texttt{Wd1} \xrightarrow{\mathsf{wrsto}} \mathtt{B} \\
\qquad \wedge\ (\neg(\mathsf{E} \Vdash \mathsf{com}(\texttt{Wd2})) \Rightarrow [\![\mathsf{E}.\mathsf{A}]\!](\mathtt{B}) = \mathtt{k} - \mathtt{a1}) \\
\qquad \wedge\ (\mathsf{E} \Vdash \mathsf{com}(\texttt{Wd2}) \Rightarrow [\![\mathsf{E}.\mathsf{A}]\!](\mathtt{B}) = \mathtt{k} - \mathtt{a1} - \mathtt{a2})
\end{array}
$$

One of the conjuncts asserts that Wd1 has written to B - a fact which cannot be deduced solely from the value of B (esp. in the presence of interference), but can be expressed as a proposition over the execution trace.

4.1 The Rely-Guarantee Judgment

The standard rely-guarantee judgment is a quintuple $\{P, R\}\, c\, \{G, Q\}$, which informally asserts that if a command c is executed in a state that satisfies its pre-condition P, provided that every interference step during the execution is contained inside the rely relation R, then the effect that each step of executing c has on the state is captured by G, and the final state of execution satisfies the post-

[5] It is important to keep in mind the distinction between high-level invariants (*e.g.,* $\mathtt{X} \geq 0$) and trace invariants (*e.g.,* $\mathbb{I} = \lambda\mathsf{E}.\, \texttt{RC(Wd1)} \wedge \texttt{SI(Wd2)}$). The former constitute *proof obligations* for the programmer, whereas the latter is a combination of transaction-specific *assumptions* about operational characteristics of the underlying data store.

Expressions in $\texttt{txn}\langle i\rangle$ $\quad \boxed{\{\mathbb{I}, P, R\}\,[e]_i\,\{G, C, Q\}}$

RG-VAR

$$\frac{\begin{array}{c}\texttt{stable}(R,\mathbb{I}) \quad \mathbb{I} \vdash \texttt{stable}(R,P) \quad \mathbb{I} \vdash \texttt{stable}(R,Q) \\ \forall \mathsf{E}, j, \eta, S, \mathsf{E}'.\ P(\mathsf{E}) \wedge j > \mathsf{maxId}(\mathsf{E.A}) \ \wedge\ S \subseteq \mathsf{E.A} \\ \wedge\ \eta = (T_i, j, \texttt{RD}(X), [\![S]\!](X)) \ \wedge\ \mathsf{E}' = \mathsf{E} \cup (\{\eta\}, S \times \{\eta\}) \\ \wedge\ \mathbb{I}(\mathsf{E}') \Rightarrow C([\![S]\!](X)) \wedge Q(\mathsf{E}') \wedge G(\mathsf{E}, \mathsf{E}')\end{array}}{\{\mathbb{I}, P, R\}\,[X]_i\,\{G, C, Q\}}$$

RG-ARITH

$$\frac{\forall v_1, v_2.\ C_1(v_1) \ \wedge\ C_2(v_2) \Rightarrow C(v_1 \pm v_2) \quad \{\mathbb{I}, P, R\}\,[e_1]_i\,\{G, C_1, Q'\} \quad \{\mathbb{I}, Q', R\}\,[e_2]_i\,\{G, C_2, Q\}}{\{\mathbb{I}, P, R\}\,[e_1 \pm e_2]_i\,\{G, C, Q\}}$$

RG-CONSEQ

$$\frac{\mathbb{I}' \Rightarrow \mathbb{I} \quad P' \Rightarrow P \quad Q \Rightarrow Q' \quad C \Rightarrow C' \quad R' \subseteq R \quad G \subseteq G' \quad \{\mathbb{I}, P, R\}\,[e]_i\,\{G, C, Q\}}{\{\mathbb{I}', P', R'\}\,[e]_i\,\{G', C', Q'\}}$$

Commands in $\texttt{txn}\langle i\rangle$ $\quad \boxed{\{\mathbb{I}, P, R\}\,[c]_i\,\{G, Q\}}$

RG-ASGN

$$\frac{\begin{array}{c}\{\mathbb{I}, P, R\}\,[e]_i\,\{G, C, Q'\} \\ \texttt{stable}(R,\mathbb{I}) \quad \mathbb{I} \vdash \texttt{stable}(R,Q') \quad \mathbb{I} \vdash \texttt{stable}(R,Q) \\ \forall v, \mathsf{E}, j, \eta, S, \mathsf{E}'.\ C(v) \ \wedge\ Q'(\mathsf{E}) \ \wedge\ j > \mathsf{maxId}(\mathsf{E.A}) \ \wedge\ S \subseteq \mathsf{E.A} \\ \wedge\ \eta = (T_i, j, \texttt{WR}(X), v) \ \wedge\ \mathsf{E}' = \mathsf{E} \cup (\{\eta\}, S \times \{\eta\}) \\ \wedge\ \mathbb{I}(\mathsf{E}') \Rightarrow Q(\mathsf{E}') \ \wedge\ G(\mathsf{E}, \mathsf{E}')\end{array}}{\{\mathbb{I}, P, R\}\,[X := e]_i\,\{G, Q\}}$$

Top-Level Commands $\quad \boxed{\{\mathbb{I}, P, R\}\,c\,\{G, Q\}}$

RG-TXN

$$\frac{\begin{array}{c}\{\mathbb{I}, P, R\}\,[c]_i\,\{G, Q'\} \\ \texttt{stable}(R,\mathbb{I}) \quad \mathbb{I} \vdash \texttt{stable}(R,Q) \quad \mathbb{I} \vdash \texttt{stable}(R,Q') \\ \forall \mathsf{E}, j, \eta, S, \mathsf{E}'.\ Q'(\mathsf{E}) \ \wedge\ j > \mathsf{maxId}(\mathsf{E.A}) \ \wedge\ S \subseteq \mathsf{E.A} \\ \eta = (T_i, j, \texttt{COMMIT}, \bot) \ \wedge\ \mathsf{E}' = \mathsf{E} \cup (\{\eta\}, S \times \{\eta\}) \\ \wedge\ \mathbb{I}(\mathsf{E}') \Rightarrow Q(\mathsf{E}') \ \wedge\ G(\mathsf{E}, \mathsf{E}')\end{array}}{\{\mathbb{I}, P, R\}\,\texttt{txn}\langle i\rangle\{c\}\,\{G, Q\}}$$

Figure 9: $\mathcal{T}$: Rely-Guarantee Rules

condition Q. The terms "execution" and "step" are defined with respect to an operational semantics, which, in our case, is parameterized on the trace invariant $\mathbb{I}$. Consequently, our rely-guarantee judgment for commands is the sextuple $\{\mathbb{I}, P, R\}\,c\,\{G, Q\}$, whose semantics differs from the quintuple in that it requires every step of the execution to additionally preserve the trace invariant $\mathbb{I}$ for the post-condition to be valid. As usual, if c is inside a transaction T_i, we write $[c]_i$. $\mathcal{T}$expressions are side-effecting (they generate RD effects), and admit interference during their evaluation. Therefore, like commands, expressions need to be reasoned about explicitly. But, unlike commands, expressions evaluate to values, and the reasoning framework should also admit assertions on such values. We therefore define a separate judgment for expressions - the septuple $\{\mathbb{I}, P, R\}\,e\,\{G, C, Q\}$. The new entrant $C : \mathbb{N} \to \mathbb{P}$ is an assertion on the return value of e.

The rules that define rely-guarantee judgment are shown in Fig. 9. Like standard rely-guarantee definitions, these definitions also require a *stability* condition, which requires pre- and post-conditions to hold despite any interference from concurrent threads (captured by R). Stability can be predicated on the assumption that interference preserves the trace invariant $\mathbb{I}$. Formally:

$$\mathbb{I} \vdash \texttt{stable}(R,P) \quad \stackrel{def}{=} \quad \forall \mathsf{E}, \mathsf{E}'.\, \mathbb{I}(\mathsf{E}) \ \wedge\ P(\mathsf{E}) \ \wedge\ R(\mathsf{E}, \mathsf{E}') \ \wedge\ \mathbb{I}(\mathsf{E}') \Rightarrow P(\mathsf{E}')$$

However, the assumption that interference preserves $\mathbb{I}$ (or, dually, $\mathbb{I}$ withstands interference) needs to be justified separately. We call this the stability requirement on $\mathbb{I}$:

$$\texttt{stable}(R,\mathbb{I}) \quad \stackrel{def}{=} \quad \forall \mathsf{E}, \mathsf{E}'.\, \mathbb{I}(\mathsf{E}) \ \wedge\ R(\mathsf{E}, \mathsf{E}') \Rightarrow \mathbb{I}(\mathsf{E}')$$

The rule RG-VAR defines the rely-guarantee judgment for shared variable reads inside a transaction T_i. It requires $\mathbb{I}$ to be stable, and pre- and post- conditions to be stable relative to $\mathbb{I}$. The quantified premise effectively requires a proof that if the abstract machine of Fig. 5 takes a step starting from an execution E that satisfies the pre-condition P, then the resultant execution E' satisfies the post-condition Q, and that the guarantee G faithfully captures the transition from E to E'. It is informative to compare this premise with the premise of the E-AUX reduction rule of Fig. 5. Similar premises also appear in the RG-ASGN and RG-TXN rules, which define rely-guarantee judgments for assignments and transactions, respectively. RG-VAR however also requires the return value ($[\![S]\!](X)$) of the read to satisfy the assertion C meant for the value. RG-ARITH defines the RG judgment for an arithmetic expression $e_1 \pm e_2$ in terms of the corresponding judgments for the constituent expressions e_1 and e_2. The quantified premise requires any value resulting from evaluating $e_1 \pm e_2$ to satisfy the assertion C, provided that e_1 and e_2 always evaluate to values that satisfy C_1 and C_2, respectively. The rules for sequential and parallel composition of commands are essentially the same as their counterparts in a standard rely-guarantee formulation and hence elided.

The RG-CONSEQ rule defines ways to strengthen or weaken relations and assertions associated with the RG judgment of transaction-bound expressions. Similar rules exist for transaction-bound and top-level commands, but are not shown.[6] As is the case with a standard rely-guarantee formulation,the rules allow the pre-condition P and the rely relation R to be strengthened, and the post-condition Q (also, C in the case of expressions) and the guarantee relation G to be weakened. The most notable aspect of the RG-CONSEQ rules is that they allow the trace invariant $\mathbb{I}$ to be strengthened. Considering that $\mathbb{I}$ captures isolation properties, this means that a program proven correct under weaker isolation levels is also correct under stronger ones. Parametricity over the trace invariant $\mathbb{I}$, combined with the ability to strengthen $\mathbb{I}$ as needed, allows our proof system to support a highly flexible proof strategy to prove programs correct over various isolation variants. For example, programmers can *define* isolation guarantees of their choice (by defining $\mathbb{I}$ appropriately) and then prove programs correct assuming the guarantees hold. The soundness of strengthening $\mathbb{I}$ ensures that a program can be safely executed on any system that offers isolation guarantees at least as strong as those assumed.

4.2 Semantics and Soundness

Definition 4.1 (Step-indexed reflexive transitive closure). For all A : Type, $R : A \to A \to \mathbb{P}$, and $n : \mathbb{N}$, the step-indexed reflexive transitive closure R^n of R is the smallest relation satisfying the following properties:

- $\forall (x : A).\, R^0(x, x)$
- $\forall (n : \mathbb{N})(x, y, z : A).\, R(x, y) \ \wedge\ R^n(y, z) \Rightarrow R^{n+1}(x, z)$

Definition 4.2 (Interleaved step and multi-step relations). An interleaved step relation interleaves thread-local reductions with interference from concurrent threads captured as the rely relation (R). It is defined thus:

$$\mathbb{I} \vdash (c, \mathsf{E}) \longrightarrow_R (c', \mathsf{E}') \quad \stackrel{def}{=} \quad \mathbb{I} \vdash (c, \mathsf{E}) \longrightarrow (c', \mathsf{E}') \ \vee\ (c' = c \ \wedge\ R(\mathsf{E}, \mathsf{E}') \ \wedge\ \mathbb{I}(\mathsf{E}'))$$

The interleaved step relation for transaction bound expressions ($[e]_i$) and commands ($[c]_i$) is defined similarly. An interleaved multi-step relation ($\longrightarrow_R^n$) is the step-indexed reflexive transitive closure of the interleaved step relation.

[6] The supplementary materials provide the complete set of rules.

Definition 4.3 (**Semantics of the RG judgment).** The semantics of the RG sextuple $\{\mathbb{I}, P, R\}\, c\, \{G, Q\}$ is defined in terms of the interleaved step relation thus:

$$\begin{aligned}\{\mathbb{I}, P, R\}\, c\, \{G, Q\} \stackrel{def}{=}\ & \forall \mathsf{E}.\ P(\mathsf{E}) \wedge \mathbb{I}(\mathsf{E}) \\ & \Rightarrow (\forall n, \mathsf{E}'.\ \mathbb{I} \vdash (c, \mathsf{E}) \longrightarrow_R^n (\mathtt{SKIP}, \mathsf{E}') \Rightarrow Q(\mathsf{E}')) \\ & \quad \wedge\ \mathtt{step\text{-}guaranteed}(\mathbb{I}, R, G, c, \mathsf{E})\end{aligned}$$

The first conjunct in the consequent is called the *Hoare consequent* since it ascribes Hoare triple semantics to an RG sextuple. The second conjunct, called the *guarantee consequent*, uses the `step-guaranteed` predicate defined below:

$$\begin{aligned}&\mathtt{step\text{-}guaranteed}(\mathbb{I}, R, G, c, \mathsf{E}) \stackrel{def}{=}\ \forall n, \mathsf{E}', c'', \mathsf{E}''. \\ &\quad \mathbb{I} \vdash (c, \mathsf{E}) \longrightarrow_R^n (c', \mathsf{E}') \wedge \mathbb{I} \vdash (c', \mathsf{E}') \longrightarrow (c'', \mathsf{E}'') \Rightarrow G(\mathsf{E}', \mathsf{E}'')\end{aligned}$$

The guarantee consequent requires G to capture the trace effect of every small-step of c, where the reduction can be interleaved by the interference (R) from concurrent threads. The semantics of the RG sextuple for transaction-bound commands ($[c]_i$) is defined similarly. Expressions, unlike commands, evaluate to a value v, and the semantics of their RG septuple ($\{\mathbb{I}, P, R\}\, [e]_i\, \{G, C, Q\}$) differs slightly in that its Hoare consequent requires the value v to satisfy the assertion C.

Note that the semantics of all RG judgments, including the judgments for transaction-bound terms, make similar demands of the guarantee relation. Given that transactions are atomic (though not isolated), it is not immediately apparent why a transaction's guarantee is required to make explicit every step of its reduction. This requirement is justified however because, in reality, a transaction's atomicity is predicated on the isolation settings of the observer. A *Read Uncommitted* transaction, for example, is permitted to observe the internal state of a transaction T even if T is claimed to execute atomically. In the interest of modular verification, the transaction must therefore make its internal state available via its guarantee relation.

Theorem 4.4 (**Soundness).** *The rely-guarantee judgments defined by the rules in Fig. 9 are sound with respect to the semantics of Definition 4.3.*[7]

In particular, if $\{\mathbb{I}, P, R\}\, c\, \{G, Q\}$ can be derived using the rules of Fig. 9, then (a) every interleaved multi-step reduction of c starting from a trace that satisfies P and $\mathbb{I}$, results in a trace that satisfies Q, and (b) the effect that every small-step of c has on the trace is contained in G. Soundness of the RG judgment for transaction-bound commands ($[c]_i$) is stated similarly. For expressions, soundness of the judgment $\{\mathbb{I}, P, R\}\, [e]_i\, \{G, C, Q\}$ also proves that e is always evaluated to a value that satisfies C.

5. Data Stores and Consistency

The operational semantics of Fig. 5 allows operations to witness arbitrary subsets of the global state, effectively mimicking the behaviour of an eventually consistent (EC) data store[8]. There are, however, data stores, such as relational databases, that provide stronger consistency guarantees than EC. Like the isolation levels of transactions, the consistency level of the underlying store also affects the semantics of a program in non-trivial ways. In this section, we demonstrate how the semantics of stronger stores with on-demand weak isolation can be captured in our operational model.

First, we observe that the consistency level of a data store can be captured by store-specific consistency constraints, along with transaction-specific isolation constraints, via the trace invariant $\mathbb{I}$. In particular, we can split $\mathbb{I}$ into two components: (1). $\mathbb{I}_s$, the store-specific invariant, and (2). $\mathbb{I}_c$, the program-specific (or, client-specific) invariant, to capture consistency and isolation constraints[9], respectively. $\mathbb{I}$ is now a conjunction: $\mathbb{I} = \lambda \mathsf{E}.\ \mathbb{I}_s(\mathsf{E}) \wedge \mathbb{I}_c(\mathsf{E})$ (often simplified to $\mathbb{I} = \mathbb{I}_s \wedge \mathbb{I}_c$). While the program-specific trace invariant ($\mathbb{I}_c$) is independent of the underlying data store, the store specific invariant ($\mathbb{I}_s$) changes from store to store depending on the consistency level. For an eventually consistent store, $\mathbb{I}_s$ is simply *true*. Stronger stores, such as those that support strong consistency, have non-trivial definitions for $\mathbb{I}_s$.

A strongly-consistent SC store guarantees a total order on all operations w.r.t. vis consistent with their chronological order. A straightforward $\mathbb{I}_s$ for this store is the SC property formalized below:

$$\begin{aligned}\mathtt{SC}(\mathsf{E}) \ =\ & \forall \eta_1, \eta_2.\ \{\eta_1, \eta_2\} \subseteq \mathsf{E.A} \wedge \mathsf{id}(\eta_1) < \mathsf{id}(\eta_2) \\ & \Rightarrow \mathsf{E} \Vdash \eta_1 \xrightarrow{\mathsf{vis}} \eta_2\end{aligned}$$

Unfortunately, $\mathbb{I}_s = \mathtt{SC}$ conflicts with all isolation specifications of Fig. 7. For instance, consider a case where $\mathbb{I}_c(\mathsf{E}) = \forall T_i.\ \mathsf{E} \Vdash \mathtt{RC}(T_i)$, a constraint that dictates all transactions execute under *Read Committed* isolation. Imagine a sample execution where η_1's transaction is not yet committed when η_2 is generated. Letting η_1 be visible to η_2 violates $\mathbb{I}_c$, whereas not letting it be visible violates $\mathbb{I}_s$. The only way to satisfy both invariants is to rule out all executions that interleave the operations of one transaction with the other, thereby enforcing serializability. In general, when $\mathbb{I}_s$ conflicts (but is not inconsistent) with $\mathbb{I}_c$, the only way to enforce both is to restrict concurrency. Clearly, this is unacceptable since it defeats the very purpose of supporting weak isolation.

In practice, relational databases resolve such conflicts by prioritizing weak isolation (thus, concurrency and performance) over strong consistency, so the execution traces do not necessarily satisfy SC. In particular, visibility constraints imposed by SC are violated iff they are found to be in conflict with the constraints imposed by a transaction's isolation level. In the context of the aforementioned example, η_1 is not made visible to η_2 because doing so would violate $\mathbb{I}_c$. However, if $\mathbb{I}_c$ is redefined to *true*, then the store makes η_1 visible to η_2 to honor its consistency commitment[10]. We generalize this approach to any $\mathbb{I}_s$ and $\mathbb{I}_c$ by defining a *maximum visibility principle* to determine an acceptable weakening of $\mathbb{I}_s$ in case of a conflict with $\mathbb{I}_c$. The principle requires the weakened consistency guarantee ($\mathbb{I}'_s$) of the store to enforce all visibility relationships imposed by the actual consistency guarantee ($\mathbb{I}_s$), unless enforcing such a relationship violates $\mathbb{I}_c$. Formally:

Definition 5.1. $\mathbb{I}'_s : \mathsf{E} \to \mathbb{P}$ is said to be a maximum visibility weakening of $\mathbb{I}_s : \mathsf{E} \to \mathbb{P}$ if and only if:

- $\mathbb{I}'_s$ is weaker than $\mathbb{I}_s$: $\forall \mathsf{E}.\ \mathbb{I}_s(\mathsf{E}) \Rightarrow \mathbb{I}'_s(\mathsf{E})$, and
- In every trace E that satisfies $\mathbb{I}_c$, and for every pair of effects η_1 and η_2 in E, if $\mathbb{I}_s(\mathsf{E})$ requires η_1 to be visible to η_2, then so does $\mathbb{I}'_s(\mathsf{E})$ unless extending E with $\mathsf{vis}(\eta_1, \eta_2)$ violates $\mathbb{I}_c$:

$$\begin{aligned}&\forall \mathsf{E}, \eta_1, \eta_2.\ \mathbb{I}_c(\mathsf{E}) \Rightarrow (\mathbb{I}_s(\mathsf{E}) \Rightarrow \mathsf{E} \Vdash \eta_1 \xrightarrow{\mathsf{vis}} \eta_2) \Rightarrow \\ &\qquad (\mathbb{I}'_s(\mathsf{E}) \Rightarrow \mathsf{E} \Vdash \eta_1 \xrightarrow{\mathsf{vis}} \eta_2 \ \vee\ \neg \mathbb{I}_c(\mathsf{E} \cup (\emptyset, \{(\eta_1, \eta_2)\})))\end{aligned}$$

Applying this principle, we can weaken SC to obtain the following store trace invariant ($\mathbb{I}_s$) for an SC store whose isolation constraints are captured by $\mathbb{I}_c$:

$$\begin{aligned}\mathbb{I}_s(\mathsf{E}) \quad =\quad & \forall \eta_1, \eta_2.\ \{\eta_1, \eta_2\}, \subseteq \mathsf{E.A} \wedge \mathsf{id}(\eta_1) < \mathsf{id}(\eta_2) \\ & \Rightarrow \mathsf{E} \Vdash \eta_1 \xrightarrow{\mathsf{vis}} \eta_2 \ \vee\ \neg \mathbb{I}_c(\mathsf{E} \cup (\emptyset, \{(\eta_1, \eta_2)\}))\end{aligned}$$

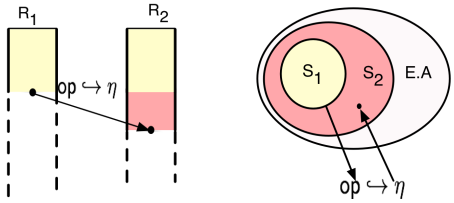
$\mathbb{I}_s$ requires a trace E to satisfy the visibility constraints of SC except in cases where they are in conflict with $\mathbb{I}_c$. Instantiating the parameter $\mathbb{I}$ with $\mathbb{I}_s \wedge \mathbb{I}_c$ in Fig. 5 results in an operational semantics that

[7] Formal proof of soundness is provided in the supplementary material.

[8] Eventual consistency guarantees that in the absence of further updates, all reads witness the same global state *eventually*. In any finite trace, however, there are no guarantees on what a read may witness.

[9] $\mathbb{I}_c$ captures the combination of transaction-specific isolation constraints.

[10] The term *recency commitment* [5] is often used in practice to capture the best-effort nature of SC.



(a) Replica model (b) Subset model

Figure 10: In the replica model, operation op generates effect η at replica R_1, which is then merged to R_2. If the *store is* CC, then R_2's state at the merge event is same or larger than R_1's state at the generation event (the difference is highlighted). In our subset model, op witnesses $S_1 \subseteq$ E.A and generates η, which is immediately added to E.A. A later operation may witness $S_2 \subseteq$ E.A, and if the *operation is* CC and $\eta \in S_2$, then it also witnesses S_1 (i.e., $S_1 \subseteq S_2$).

describes SC stores with on-demand weak isolation, such as provided by off-the-shelf relational databases. Similar approaches can be employed to capture the semantics of other kinds of stores, such as a causally consistent (CC) data store, as specific instantiations of our operational semantics[11].

5.1 Weakly Consistent Replication

Reasoning under weakly-consistent replication has received special attention in recent work [30]. Our operational semantics and proof system are general enough to admit replication as a special case of our formulation, as we explain below.

The primary challenge in this setting is to ensure that the assumptions made and guarantees enforced by an operation at one replica carry over to other replicas that merge their effects, thus preserving the overall integrity of the system. In prior work [30, 34], this challenge is partly addressed by imposing restrictions on how various replica states differ, i.e., by fixing a system model with a stronger baseline consistency (CC) than EC. This unfortunately restricts the reasoning approach from being applied to data stores (e.g., [46, 49]) that provide guarantees weaker than causal consistency, such as causal visibility or read-my-writes [52]. Causal consistency is not baseline consistency in these stores because it is not *highly available* [5].

Notably, our view of replication does not explicitly involve replicas. Fig. 10 contrasts our model of weakly-consistent replication with a conventional replica-based model. Under our model, the notion of a replica is subsumed by the concept of visibility; a replica is defined by the subset (S) of global state (E.A) that an operation witnesses. Constraints over replica states therefore manifest as constraints over the visibility relation. For example, instead of requiring the store to be causally consistent, an operation can witness a causally consistent subset of the state; such demands can be made via the trace invariant $\mathbb{I}$. For a pre-condition (P) of the operation to be useful, it has to be an assertion over every causally consistent subset of the global state. Since any replica that eventually executes the operation has to expose one such subset (S), the pre-condition is guaranteed to hold regardless of the replica. There is however one problem with this explanation - by considering subsets of just one global state, it ignores the fact that the global state (hence, the replica states) change during the execution of the operation. To account for such changes, we might choose to distinguish between an effect generation event at one replica r_1 and an effect merge event at replica r_2, requiring that *non-conflicting* operations execute between these two events at r_2, and that they preserve certain invariants [30]. Instead, our framework folds all such machinery into a stability condition predicated on $\mathbb{I}$ (§4.1). Since any change to the global state during the execution of the operation is an interference, and P is required to be stable with respect to any such interference, it follows that P is valid on every replica, thus ensuring that assumptions made at a generation event is also valid at the merge event.

[11] Details of the CC instantiation are given in the supplementary material.

$$P : \{\neg\mathsf{com}(\mathtt{Wd1}) \ \wedge \ \neg\mathsf{com}(\mathtt{Wd2}) \ \wedge \ \mathtt{B} = \mathtt{k}\}$$

$$P_1 : \{\neg\mathsf{com}(\mathtt{Wd1}) \ \wedge \ (\neg\mathsf{com}(\mathtt{Wd2}) \Rightarrow \mathtt{B} = \mathtt{k}) \ \wedge \ \mathsf{com}(\mathtt{Wd2}) \Rightarrow \mathtt{Wd2} \xrightarrow{\mathsf{wrsto}} \mathtt{B} \wedge \mathtt{B} = \mathtt{k} - \mathtt{a2}\}$$

$\mathtt{txn}\langle\mathtt{Wd1}\rangle\{$

$$\vdots$$

$$\phi_4 : \{\neg\mathsf{com}(\mathtt{Wd2}) \Rightarrow \mathtt{B} = \mathtt{k} \wedge \mathtt{v3} = \mathtt{k} - \mathtt{a1} \ \wedge \ (\mathsf{com}(\mathtt{Wd2}) \Rightarrow \mathtt{Wd2} \xrightarrow{\mathsf{wrsto}} \mathtt{B}) \ \wedge \ \neg\mathsf{com}(\mathtt{Wd1}) \ \wedge \ \mathsf{com}(\mathtt{Wd2}) \wedge \mathtt{Wd2} \xrightarrow{\mathsf{vis}} \mathtt{Wd1} \Rightarrow \mathtt{B} = \mathtt{k} - \mathtt{a2} \wedge \mathtt{v3} = \mathtt{k} - \mathtt{a2} - \mathtt{a1}\}$$

$\mathtt{B} := \mathtt{v3}$

$$\phi_5 : \{(\neg\mathsf{com}(\mathtt{Wd2}) \Rightarrow \mathtt{B} = \mathtt{k} - \mathtt{a1}) \ \wedge \ \neg\mathsf{com}(\mathtt{Wd1}) \ \wedge \ \mathsf{com}(\mathtt{Wd2}) \Rightarrow \mathtt{Wd2} \xrightarrow{\mathsf{wrsto}} \mathtt{B} \wedge \mathtt{B} = \mathtt{k} - \mathtt{a1} - \mathtt{a2}\}$$

$$\vdots$$

$\}$

$$Q_1 : \{(\neg\mathsf{com}(\mathtt{Wd2}) \Rightarrow \mathtt{B} = \mathtt{k} - \mathtt{a1}) \ \wedge \ \mathsf{com}(\mathtt{Wd1}) \ \wedge \ \mathsf{com}(\mathtt{Wd2}) \Rightarrow \mathtt{B} = \mathtt{k} - \mathtt{a1} - \mathtt{a2}\}$$

$$Q : \{\mathsf{com}(\mathtt{Wd1}) \ \wedge \ \mathsf{com}(\mathtt{Wd2}) \ \wedge \ \mathtt{B} = \mathtt{k} - \mathtt{a1} - \mathtt{a2}\}$$

Figure 11: Wd1 transaction decorated with assertions

5.2 Example

We now consider the proof of the example in Fig. 1 in greater detail. Both withdraw transactions are run at SI isolation, hence $\mathbb{I}_c$ is λE. E $\Vdash$ SI(Wd1) $\wedge$ E $\Vdash$ SI(Wd2). We assume an SC store, such as a conventional RDBMS, whose store-specific trace invariant ($\mathbb{I}_s$) is the maximum visibility weakening of SC w.r.t. $\mathbb{I}_c$, as described in §5. In concrete terms, this means that an effect added to E.A by the current thread or a concurrent thread immediately becomes visible to the subsequent operations unless we produce a witness that such visibility violates $\mathbb{I}_c$. As usual, $\mathbb{I}$ is $\mathbb{I}_s \ \wedge \ \mathbb{I}_c$. Note that the language of our specifications is a decidable subset of the first-order logic. In particular, our specifications and rely-guarantee rules are carefully crafted to avoid generating proof obligations with transitive closures and quantification over infinite domains. The domain of assertions is often linear arithmetic, which is also decidable. Consequently, the generated proof obligations can often be discharged via an SMT solver.

Decorated parts of Wd1 relevant to the discussion are shown in Fig. 11. Wd2 is similar and not shown. Temporary name v3 has been introduced to capture the result of the RHS expression B-a1. Assertions implicitly refer to the current execution (E), just as Hoare triples implicitly refer to the current state. The context for propositions and the state for shared variables is also the implicit, i.e., we write ψ instead of E $\Vdash \psi$ and B = k instead of $[\![$E.A$]\!]$(B) = k. The proposition k $\geq$ a1+a2 remains an invariant, hence elided. The pre-condition P_1 of Wd1 accounts for the possibility of Wd2 committing before Wd1, writing k-a2 to B. Since neither Wd1 nor Wd2 are committed at the beginning, the pre-condition P of the program is extended with $\neg$com(Wd1) $\wedge$ $\neg$com(Wd2), from which P_1 follows. The post-condition Q_1 of Wd1 asserts different values for B depending on whether or not Wd2 has already committed. It also asserts that Wd1 has committed. The post-condition Q_2 of Wd2 (not

shown) is similar, but with the roles of Wd1 and Wd2 exchanged. The post-condition Q follows from $Q_1 \wedge Q_2$, hence valid. It remains to show that Wd1 satisfies its specification (Wd2 must also satisfy its spec, but the proof is similar).

First, we focus on the sequential aspect of the proof and show that the assertions that decorate Wd1 are indeed valid. We consider the triple associated with the assignment to B for illustration. The pre-condition ϕ_4 asserts different values for B and v3 depending on whether or not Wd2 is committed. Inside Wd1, $\mathsf{com}(\texttt{Wd2})$ may mean that either Wd2 happened before Wd1 (hence can be visible to all of Wd1), or that it committed concurrently with Wd1 (hence cannot be visible to all of Wd2). Nonetheless, ϕ_4 only considers the case when $\texttt{Wd2} \xrightarrow{\mathsf{vis}} \texttt{Wd1}$. As we show below, this is sufficient to prove the post-condition ϕ_5, which appears to make a stronger claim about the value of B. The proof follows from the RG rule RG-ASGN for assignments, which allows us to easily conclude the following about the execution state after the assignment:

$$\neg\mathsf{com}(\texttt{Wd1}) \wedge \texttt{Wd1} \xrightarrow{\mathsf{wrsto}} \texttt{B} \wedge (\neg\mathsf{com}(\texttt{Wd2}) \Rightarrow \texttt{B} = \texttt{k} - \texttt{a1}) \wedge$$
$$(\mathsf{com}(\texttt{Wd2}) \Rightarrow \texttt{Wd2} \xrightarrow{\mathsf{wrsto}} \texttt{B}) \wedge$$
$$(\mathsf{com}(\texttt{Wd2}) \wedge \texttt{Wd2} \xrightarrow{\mathsf{vis}} \texttt{Wd1} \Rightarrow \texttt{B} = \texttt{k} - \texttt{a2} \wedge \texttt{B} = \texttt{k} - \texttt{a2} - \texttt{a1})$$

The rule also lets us assume that the execution satisfies trace invariant ($\mathbb{I}$), which asserts SI for both transactions. Since both transactions now write to B, SI requires either $\texttt{Wd1} \xrightarrow{\mathsf{vis}} \texttt{Wd2}$, or $\texttt{Wd2} \xrightarrow{\mathsf{vis}} \texttt{Wd1}$. Since Wd1 is not yet committed ($\neg\mathsf{com}(\texttt{Wd1})$), SI, which subsumes RC and requires only committed transactions be visible, prohibits $\texttt{Wd1} \xrightarrow{\mathsf{vis}} \texttt{Wd2}$, allowing us the deduce $\texttt{Wd2} \xrightarrow{\mathsf{vis}} \texttt{Wd1}$. This lets us derive $\mathsf{com}(\texttt{Wd2}) \Rightarrow \texttt{B} = \texttt{k} - \texttt{a2} \wedge \texttt{B} = \texttt{k} - \texttt{a2} - \texttt{a1}$, allowing us to prove the post-condition ϕ_5.

The second part of the proof is to show that assertions are stable despite the interference from the concurrent thread executing Wd2. We formalize such interference as a rely relation that describes the semantics of Wd2 at the right level of abstraction. In particular, we would like to capture two important facts about Wd2:

- If Wd2 commits, then it should have already written to B. This fact is needed to show that Wd2 conflicts with Wd1, so that SI(Wd1) property can be exploited to deduce $\texttt{Wd2} \xrightarrow{\mathsf{vis}} \texttt{Wd1}$ from $\mathsf{com}(\texttt{Wd2})$ in the sequential proof.
- The value written to B is either $\texttt{k} - \texttt{a1} - \texttt{a2}$ or $\texttt{k} - \texttt{a1}$ depending on whether or not Wd1 has already committed. This is clearly necessary to deduce the value of B before and after Wd1 commits.

The resultant rely relation (R_1) is shown below[12]:

$$\begin{aligned} R_1 \quad = \quad \{(\mathsf{E}, \mathsf{E}') \mid \neg\mathsf{E} \Vdash \mathsf{com}(\texttt{Wd1}) \wedge (\mathsf{E}' - \mathsf{E}) \Vdash \mathsf{com}(\texttt{Wd2}) \\ \Rightarrow \texttt{B} = \texttt{k} - \texttt{a2} \wedge \mathsf{E}' \Vdash \texttt{Wd2} \xrightarrow{\mathsf{wrsto}} \texttt{B} \\ \wedge \mathsf{E} \Vdash \mathsf{com}(\texttt{Wd1}) \wedge (\mathsf{E}' - \mathsf{E}) \Vdash \mathsf{com}(\texttt{Wd2}) \\ \Rightarrow \texttt{B} = \texttt{k} - \texttt{a1} - \texttt{a2} \wedge \mathsf{E}' \Vdash \texttt{Wd2} \xrightarrow{\mathsf{wrsto}} \texttt{B} \\ \wedge \forall(\eta \in \mathsf{E}'.\mathsf{A} - \mathsf{E}.\mathsf{A}).\ \mathsf{txn}(\eta) = \texttt{Wd2}\} \end{aligned}$$

For the sake of completeness, we also explicitly represent the fact that all the effects added by R_1 are from Wd2. This is needed to prove that an interference from Wd2 does not affect the local state of Wd1.

The stability for ϕ_4 w.r.t. R_1 is trivial to establish: if R_1 does not commit Wd2, then the value of B is unaffected. If it commits Wd2, then since Wd1 had already performed (read) operations that cannot witness Wd2, we have $\texttt{Wd2} \not\xrightarrow{\mathsf{vis}} \texttt{Wd1}$, and since ϕ_4 only makes assertions under the case $\texttt{Wd2} \xrightarrow{\mathsf{vis}} \texttt{Wd1}$, it is stable. For ϕ_5, the SI condition is used to show that any interference from Wd2 is invalid. The proof proceeds on the same lines as the proof discussed above, and is also discussed in detail in§2.

The final aspect of the proof is to show that any interference from Wd1 is contained in its guarantee relation G_1, which is also the rely relation R_2 for Wd2. Since the implementation of Wd2 is same as Wd1's, R_2 (hence G_1) is same as R_1 shown above, but with Wd2 and Wd1 interchanged. Continuing our focus on the assignment statement, the guarantee proof requires us to show that if executing the assignment takes the execution from E to E', such that E' satisfies $\mathbb{I}$, then $(\mathsf{E}, \mathsf{E}') \in G_1$. The proof follows from the RG-ASGN rule, which establishes that (a). the assignment statement does not commit Wd2, and (b). it adds a write effect η such that $\mathsf{txn}(\eta) = \texttt{Wd1}$. The proofs for other triples of Wd1 can also be constructed similarly. The combination of correctness proofs for Wd1 and Wd2 yields a correctness proof for the program.

6. Case Studies

We present two case studies below that demonstrate the applicability of our framework and its practical utility. Both examples target Ruby-on-Rails applications, which are commonly used to interface applications to database systems. To analyze these, and other similar open-source applications, we have developed a symbolic execution engine that compiles a Rails program to an abstract program in an extended version of $\mathcal{T}$, along with some proof automation support.[13] The focus of our formal verification efforts are such translated $\mathcal{T}$ programs.

A video view counter. Our first application is a Rails implementation of a counter to count the number of views a video has garnered on a video-sharing website like YouTube. The application stores the video count in a MySQL database, and supports a read operation (a database read transaction), and an increment operation (a read-increment-write sequence wrapped in a transaction). The application is expected to preserve a *monotonicity invariant*, namely that the view count should never appear to be decreasing. However, the default *Repeatable Read* isolation level of MySQL leads to anomalous executions - with 32 concurrent writers performing single increments, and one reader issuing a constant stream of reads, we observed 12 violations of monotonicity in 10 rounds. We subsequently instantiated our reasoning framework for an SC store with ANSI SQL isolation levels, and spent 5 man hours determining appropriate isolation levels needed to enforce the invariant. In particular, we were able to formally prove that to preserve the invariant, writers need to execute using *Serializable* isolation, while readers can execute using *Read Uncommitted*. Repeating the experiments with this configuration led to no violations, as expected. While executing all transactions at SER also prevents violations, it led to an additional 113% increase in the latency of writes averaged over 5 rounds with 32 concurrent writers. We repeated the experiments with the same configuration on Postgres, and found no violations. However, Postgres' isolation levels [40] are stronger than ANSI SQL's in non-trivial ways. We therefore formalized Postgres' guarantees and repeated relevant parts of the proof to discover that Postgres' *Repeatable Read* is in fact sufficient for writers - an observation further validated through experiments. As usual, using SER for all transactions also suffices, but on Postgres results in an additional 47% increase in write latency.

To understand the effect of changing store consistency levels, we re-implemented the counter as an op-based *replicated data type* (RDT) [17, 44] on top of Cassandra *bolted-on* [7] with CC and SC consistency levels (both on-demand; EC is default), and

[12] We write $(\mathsf{E}' - \mathsf{E}) \Vdash \mathsf{com}(\texttt{Wd2})$ to express the property that Wd2 is not committed in E, but committed in E'.

[13] Details can be found in the supplementary material. Source code of our tools, applications and experiments, is also provided in the (non-anonymous) supplementary.

ANSI SQL isolation levels. Keeping the consistency level at EC, we then repeated our experiments with committed reads and serializable increments - a combination which was proved to preserve monotonicity on an SC store, but nonetheless observed monotonicity violations. Under this instantiation, it is impossible to derive the `MonotonicVis` (§3.1) property needed to show that successive reads in a session witness monotonically increasing state. However, assuming CC for reads allowed us to complete the proof, and recover the monotonicity invariant. In fact, we were able to complete the proof even after relaxing the serializability requirement for increments, just by changing the definition of the $[\![S]\!](X)$ (§3) to return the value of the largest write-to-X instead of the latest write-to-X in S. Thus, if the reads chose the largest-valued write effect, instead of the latest write effect they witness, then counter increments need not be serializable for reads to witness a monotonically increasing count.[14] We implemented this version of the counter as an op-based RDT, which uses CC reads and EC increments, both un-isolated (i.e., *Read Uncommitted*), and obtained a 52% improvement in overall latency compared to the serializable original.

Microblog. Our second set of experiments focused on a Twitter-like microblogging application taken from [32]. An important invariant that must be preserved is referential integrity between microposts (tweets) and authors. The application enforces this invariant entirely in user space by relying on database-backed transactions (this behaviour is typical of Rails applications [9]). For instance, the deletion of a user is carried out in a transaction that also includes the deletion of microposts authored by the user. However, it is possible for concurrent observers to nonetheless witness invariant violations. To quantify the likelihood of such anomalies, we first populated a Postgres database with 1000 user accounts, in which each user is associated with 50 microposts and 20 followers, chosen uniformly at random. We then constructed experiments with 64 concurrent clients performing deletions and timeline reads of randomly chosen users in a 1:7 ratio, and witnessed 250 violations of referential integrity (the isolation level was left at the default RC level for these experiments). To determine appropriate (correct) isolation levels, we then instantiated our framework for an SC store with various ANSI SQL isolation levels for user deletions and timeline reads, and obtained a proof that RC is sufficient for deletions, whereas reads require RR. The proof is immediately applicable for MySQL, which implements ANSI SQL isolation levels. Repeating our experiments for MySQL with this configuration resulted in no violations (as expected), while yielding a 38% reduction in latency compared to an SER configuration. Since Postgres' isolation levels are stronger than the corresponding ANSI levels, the RG-CONSEQ rule (Fig. 9) lets us immediately deduce that the same choice of isolation levels also work on Postgres.

Clearly, there exists significant diversity among the weak consistency and isolation variants implemented on various real-world data stores. More such variants have been developed in recent years to meet the need for high availability (*e.g.*, *Parallel Snapshot Isolation* [47] (2011) or *Non-Monotonic Snapshot Isolation* [3] (2013)). As more commercial weakly-consistent stores adopt various kinds of transactional semantics, like Cassandra's recently introduced *lightweight transactions*, it is reasonable to expect new isolation definitions to continue to be proposed. While it may be possible to carefully engineer a reasoning framework for each combination of consistency and isolation, such a strategy would lead to multiple semantic definitions and proof systems with no obvious way to compare and relate them. As illustrated here, having a parameterized semantics and a proof framework built on top of it allows us to support these variants as distinct, yet comparable, instantiations.

[14] The trade-off is that the view count is now no longer accurate, although it grows monotonically.

7. Related Work

Specifying weak isolation. Adya [1] specifies several weak isolation levels in terms of *dependency graphs* between transactions, and the kinds of dependencies that are forbidden in each case. The operational nature of Adya's specifications make them suitable for runtime monitoring and anomaly detection [18, 41, 53], whereas the declarative nature of our specifications make them suitable for formal reasoning about program behaviour. Sivaramakrishnan *et al.* [46] specify isolation levels declaratively as trace well-formedness conditions, but their specifications implicitly assume a complete trace with only committed transactions, making it difficult to reason about a program as it builds the trace. Cerone *et al.* [20] specify isolation levels with atomic visibility, but their specifications are also for complete traces. Like ours, all the aforementioned specification frameworks use the vocabulary introduced in [17]. However, none of them are equipped with a reasoning framework that can use such specifications to verify programs under weak isolation.

Reasoning under weak isolation In [25], Fekete *et al.* propose a theory to characterize non-serializable executions that arise under SI. Fekete [24] also proposes an algorithm that allocates either SI or SER isolation levels to transactions while guaranteeing serializability. In [19], Cerone *et al.* improve on Adya's SI specification and use it to derive a static analysis that determines the safety of substituting SI with a weaker variant called *Parallel Snapshot Isolation* [47]. These efforts focus on establishing the equivalence of executions between a pair of isolation levels, without taking application invariants into account. Bernstein *et al.* [13] propose informal semantic conditions to ensure the satisfaction of application invariants under weaker isolation levels. All these techniques are tailor-made for a finite set of well-understood isolation levels (rooted in [12]) under a pre-defined store consistency model.

Reasoning under weak consistency There have been several recent proposals to reason about programs executing under weak consistency [2, 8, 10, 30, 35, 36]. All of them assume a system model that offers a choice between a *coordination-free* weak consistency level (*e.g.*, eventual consistency [2, 8, 10, 35, 36]) or causal consistency [30, 34]). All these efforts involve proving that atomic and fully isolated operations preserve application invariants when executed under these consistency levels. In contrast, we admit weakly-isolated transactions, and our system model accepts *specifications* of consistency and isolation levels drawn from an expressive logic. Gotsman *et al.* [30] adapt *Parallel Snapshot Isolation* to a transaction-less setting by interpreting it as a consistency level that serializes writes to objects; a dedicated proof rule is developed to help prove prove program invariants hold under this model. By parameterizing our proof system over a gamut of weak isolation specifications, we avoid the need to define a separate proof rule for each new isolation level we may encounter.

Reasoning under relaxed memory The reasoning mechanisms used to describe and prove properties about weakly-isolated transactions bear some resemblance to those used to formalize relaxed memory behaviour [11]. Ridge [42] generalizes rely-guarantee reasoning to the x86-TSO memory model. Likewise, Vafeiadis *et al.* [51] generalize concurrent separation logic (CSL) [50] to the C11 relaxed memory model. Ferreira *et al.* [27] propose a parameterized operational semantics for relaxed memory models, but the parameterization is over a relation between relaxed memory programs and related SC programs. Demange *et al.* [22] present a *buffered memory model* for Java that defines an axiomatic definition for the JMM in terms of memory reorderings, and an operational instantiation consistent with the TSO memory model.

References

[1] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Cambridge, MA, USA, 1999. AAI0800775.

[2] Peter Alvaro, Neil Conway, Joe Hellerstein, and William R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260, 2011.

[3] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-monotonic Snapshot Isolation: Scalable and Strong Consistency for Geo-replicated Transactional Systems. In *Proceedings of the 2013 IEEE 32Nd International Symposium on Reliable Distributed Systems*, SRDS '13, pages 163–172, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-5115-9. doi: 10.1109/SRDS.2013.25. URL `http://dx.doi.org/10.1109/SRDS.2013.25`.

[4] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1185–1196, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-5. doi: 10.1145/2463676.2465296. URL `http://doi.acm.org/10.1145/2463676.2465296`.

[5] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly Available Transactions: Virtues and Limitations. *PVLDB*, 7(3):181–192, 2013.

[6] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. HAT, Not CAP: Towards Highly Available Transactions. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 24–24, Berkeley, CA, USA, 2013. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2490483.2490507`.

[7] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-5. doi: 10.1145/2463676.2465279.

[8] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.*, 8(3):185–196, November 2014. ISSN 2150-8097. doi: 10.14778/2735508.2735509. URL `http://dx.doi.org/10.14778/2735508.2735509`.

[9] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1327–1342, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2737784. URL `http://doi.acm.org/10.1145/2723372.2737784`.

[10] Valter Balegas, Nuno Preguiça, Rodrigo Rodrigues, Sérgio Duarte, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. Putting the Consistency back into Eventual Consistency. In *Proceedings of the Tenth European Conference on Computer System*, EuroSys '15, Bordeaux, France, 2015. URL `http://lip6.fr/Marc.Shapiro/papers/putting-consistency-back-EuroSys-2015.pdf`.

[11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 55–66, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926394. URL `http://doi.acm.org/10.1145/1926385.1926394`.

[12] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM. ISBN 0-89791-731-6. doi: 10.1145/223784.223785.

[13] Arthur J. Bernstein, Philip M. Lewis, and Shiyong Lu. Semantic Conditions for Correctness at Different Isolation Levels. In *Proceedings of the 16th International Conference on Data Engineering*, ICDE '00, pages 57–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0506-6. URL `http://dl.acm.org/citation.cfm?id=846219.847381`.

[14] Philip A. Bernstein and Sudipto Das. Rethinking Eventual Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 923–928, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-5. doi: 10.1145/2463676.2465339. URL `http://doi.acm.org/10.1145/2463676.2465339`.

[15] Philip A. Bernstein and Nathan Goodman. Multiversion Concurrency Control - Theory and Algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, December 1983. ISSN 0362-5915. doi: 10.1145/319996.319998. URL `http://doi.acm.org/10.1145/319996.319998`.

[16] Bitcoin Bug. How I Stole Roughly 100 BTC From an Exchange and How I Could Have Stolen More!, 2016. URL `{https://goo.gl/4SqaP2}`. Accessed: 2016-06-4 13:22:00.

[17] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535848.

[18] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable Isolation for Snapshot Databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 729–738, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376690. URL `http://doi.acm.org/10.1145/1376616.1376690`.

[19] Andrea Cerone and Alexey Gotsman. Analysing Snapshot Isolation. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, To appear.

[20] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A Framework for Transactional Consistency Models with Atomic Visibility. In Luca Aceto and David de Frutos Escrig, editors, *26th International Conference on Concurrency Theory (CONCUR 2015)*, volume 42 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 58–71, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-91-0. doi: http://dx.doi.org/10.4230/LIPIcs.CONCUR.2015.58. URL `http://drops.dagstuhl.de/opus/volltexte/2015/5375`.

[21] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in a Partitioned Network: A Survey. *ACM Comput. Surv.*, 17(3):341–370, September 1985. ISSN 0360-0300. doi: 10.1145/5505.5508. URL `http://doi.acm.org/10.1145/5505.5508`.

[22] Delphine Demange, Vincent Laporte, Lei Zhao, Suresh Jagannathan, David Pichardie, and Jan Vitek. Plan B: A Buffered Memory Model for Java. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 329–342, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429110. URL `http://doi.acm.org/10.1145/2429069.2429110`.

[23] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM*, 19(11):624–633, November 1976. ISSN 0001-0782. doi: 10.1145/360363.360369. URL `http://doi.acm.org/10.1145/360363.360369`.

[24] Alan Fekete. Allocating Isolation Levels to Transactions. In *Proceedings of the Twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '05, pages 206–215, New York, NY, USA, 2005. ACM. ISBN 1-59593-062-0. doi: 10.1145/1065167.1065193. URL `http://doi.acm.org/10.1145/1065167.1065193`.

[25] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making Snapshot Isolation Serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005. ISSN 0362-5915.

doi: 10.1145/1071610.1071615. URL http://doi.acm.org/10.1145/1071610.1071615.

[26] Alan Fekete, Shirley N. Goldrei, and Jorge Pérez Asenjo. Quantifying Isolation Anomalies. *Proc. VLDB Endow.*, 2(1):467–478, August 2009. ISSN 2150-8097. doi: 10.14778/1687627.1687681. URL http://dx.doi.org/10.14778/1687627.1687681.

[27] Rodrigo Ferreira, Xinyu Feng, and Zhong Shao. Parameterized Memory Models and Concurrent Separation Logic. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP'10, pages 267–286, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-11956-5, 978-3-642-11956-9. doi: 10.1007/978-3-642-11957-6_15. URL http://dx.doi.org/10.1007/978-3-642-11957-6_15.

[28] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008. ISBN 9780131873254.

[29] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, June 2002. ISSN 0163-5700. doi: 10.1145/564585.564601.

[30] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm Strong Enough: Reasoning About Consistency Choices in Distributed Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 371–384, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837625. URL http://doi.acm.org/10.1145/2837614.2837625.

[31] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. pages 365–394, 1976.

[32] Michael Hartl. *The Ruby On Rails Tutorial*. Addison-Wesley, 3 edition, 2015. ISBN 978-0134077703.

[33] C. B. Jones. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, October 1983. ISSN 0164-0925. doi: 10.1145/69575.69577. URL http://doi.acm.org/10.1145/69575.69577.

[34] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified Causally Consistent Distributed Key-value Stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 357–370, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837622. URL http://doi.acm.org/10.1145/2837614.2837622.

[35] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387906.

[36] Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the Choice of Consistency Levels in Replicated Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 281–292, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-10-2. URL http://dl.acm.org/citation.cfm?id=2643634.2643664.

[37] MySQL. Transaction Isolation Levels, 2016. URL https://dev.mysql.com/doc/refman/5.6/en/innodb-transaction-isolation-levels.html. Accessed: 2016-07-1 10:00:00.

[38] Oracle. Data Concurrency and Consistency, 2016. URL https://docs.oracle.com/cd/B28359_01/server.111/b28318/consist.htm. Accessed: 2016-07-1 10:00:00.

[39] Poloniex Bug. BTC Stolen from Poloniex, 2016. URL {https://bitcointalk.org/index.php?topic=499580}. Accessed: 2016-06-4 13:22:00.

[40] Postgres. Transaction Isolation, 2016. URL https://www.postgresql.org/docs/9.1/static/transaction-iso.html. Accessed: 2016-07-1 10:00:00.

[41] Stephen Revilak, Patrick O'Neil, and Elizabeth O'Neil. Precisely Serializable Snapshot Isolation (PSSI). In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 482–493, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4244-8959-6. doi: 10.1109/ICDE.2011.5767853. URL http://dx.doi.org/10.1109/ICDE.2011.5767853.

[42] Tom Ridge. A Rely-Guarantee Proof System for x86-TSO. In *Proceedings of the Third International Conference on Verified Software: Theories, Tools, Experiments*, VSTTE'10, pages 55–70, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-15056-X, 978-3-642-15056-2. URL http://dl.acm.org/citation.cfm?id=1884866.1884873.

[43] SciMed Bug. Avoid Race Conditions that Violate Uniqueness Validation - Rails, 2016. URL http://goo.gl/0QhMQj. Accessed: 2016-06-4 13:22:00.

[44] Marc Shapiro, Nuno Preguia, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In Xavier Dfago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-24549-7. doi: 10.1007/978-3-642-24550-3_29.

[45] Dennis Shasha and Philippe Bonnet. *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. ISBN 1-55860-753-6.

[46] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 413–424, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737981. URL http://doi.acm.org/10.1145/2737924.2737981.

[47] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional Storage for Geo-replicated Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043592. URL http://doi.acm.org/10.1145/2043556.2043592.

[48] Starbucks Bug. Hacking Starbucks for unlimited coffee, 2016. URL http://sakurity.com/blog/2015/05/21/starbucks.html. Accessed: 2016-06-4 13:21:00.

[49] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM. ISBN 0-89791-715-4. doi: 10.1145/224056.224070. URL http://doi.acm.org/10.1145/224056.224070.

[50] Viktor Vafeiadis. Concurrent Separation Logic and Operational Semantics. *Electron. Notes Theor. Comput. Sci.*, 276:335–351, September 2011. ISSN 1571-0661. doi: 10.1016/j.entcs.2011.09.029. URL http://dx.doi.org/10.1016/j.entcs.2011.09.029.

[51] Viktor Vafeiadis and Chinmay Narayan. 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*, OOPSLA '13, pages 867–884, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509532. URL http://doi.acm.org/10.1145/2509136.2509532.

[52] Paolo Viotti and Marko Vukolic. Consistency in Non-Transactional Distributed Storage Systems. *CoRR*, abs/1512.00168, 2015. URL http://arxiv.org/abs/1512.00168.

[53] Kamal Zellag and Bettina Kemme. Consistency Anomalies in Multi-tier Architectures: Automatic Detection and Prevention. *The VLDB Journal*, 23(1):147–172, February 2014. ISSN 1066-8888. doi: 10.1007/s00778-013-0318-x.