View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

# A Framework for Transactional Consistency Models with Atomic Visibility

**Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman**

**IMDEA Software Institute, Madrid, Spain**

## Abstract

Modern distributed systems often rely on databases that achieve scalability by providing only weak guarantees about the consistency of distributed transaction processing. The semantics of programs interacting with such a database depends on its consistency model, defining these guarantees. Unfortunately, consistency models are usually stated informally or using disparate formalisms, often tied to the database internals. To deal with this problem, we propose a framework for specifying a variety of consistency models for transactions uniformly and declaratively. Our specifications are given in the style of weak memory models, using structures of events and relations on them. The specifications are particularly concise because they exploit the property of atomic visibility guaranteed by many consistency models: either all or none of the updates by a transaction can be visible to another one. This allows the specifications to abstract from individual events inside transactions. We illustrate the use of our framework by specifying several existing consistency models. To validate our specifications, we prove that they are equivalent to alternative operational ones, given as algorithms closer to actual implementations. Our work provides a rigorous foundation for developing the metatheory of the novel form of concurrency arising in weakly consistent large-scale databases.

**1998 ACM Subject Classification** C.2.4 Distributed Systems

**Keywords and phrases** Replication, Consistency models, Transactions

**Digital Object Identifier** 10.4230/LIPIcs.CONCUR.2015.58

## 1 Introduction

To achieve availability and scalability, modern distributed systems often rely on *replicated databases*, which maintain multiple *replicas* of shared data. The database clients can execute transactions on the data at any of the replicas, which communicate changes to each other using message passing. For example, large-scale Internet services use data replicas in geographically distinct locations, and applications for mobile devices keep replicas locally as well as in the cloud to support offline use. Ideally, we want the concurrent and distributed processing in a replicated database to be transparent, as formalised by the classical notion of serialisability [20]: the database behaves as if it executed transactions serially on a non-replicated copy of the data. However, achieving this ideal requires extensive coordination between replicas, which slows down the database and even makes it unavailable if network connections between replicas fail [1]. For this reason, nowadays replicated databases often provide weaker consistency guarantees, which allow non-serialisable behaviours, called *anomalies*. For example, consider the following program issuing transactions concurrently:

$$\begin{array}{c}\mathsf{txn}\,\{\,x.\mathtt{write}(\mathit{post});\ y.\mathtt{write}(\mathit{empty})\,\}\ \parallel\ \mathsf{txn}\,\{\,u = x.\mathtt{read}();\ y.\mathtt{write}(\mathit{comment})\,\}\\ \parallel\ \mathsf{txn}\,\{\,v = x.\mathtt{read}();\ w = y.\mathtt{read}()\,\}\end{array} \tag{1}$$

where $x, y$ are database objects and $u, v, w$ local variables. In some databases the above program can execute so that the last transaction observes the *comment*, but not the *post*:

© Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman;
licensed under Creative Commons License CC-BY
26th International Conference on Concurrency Theory (CONCUR 2015).
Editors: Luca Aceto and David de Frutos Escrig; pp. 58–71

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$u = post$, $v = empty$, $w = comment$. This result cannot be obtained by executing the three transactions in any sequence and, hence, is not serialisable. In an implementation it may arise if the first two transactions are executed at a replica $r$, and the third one at another replica $r'$, and the messages carrying the updates by the first two transactions arrive to $r'$ out of order.

The semantics of programs interacting with a replicated database thus depends on its *consistency model*, restricting the anomalies it can exhibit and, as a consequence, the possible performance optimisations in its implementation. Recent years have seen a plethora of proposals of consistency models for replicated databases [6, 19, 12, 24, 21, 13, 4] that make different trade-offs between consistency and performance. Unfortunately, these subtle models are usually specified informally or using disparate formalisms, often tied to database internals. Whereas some progress in formalising the consistency models has been recently made for replicated databases without transactions [11, 12], the situation is worse for databases providing these. The lack of a uniform specification formalism represents a major hurdle in developing the metatheory of the novel form of concurrency arising in weakly consistent replicated databases and, in particular, methods for formal reasoning about application programs using them.

To deal with this problem, we propose a framework to uniformly specify a variety of modern transactional consistency models. We apply the framework to specify six existing consistency models for replicated databases; the results are summarised in Figure 1, page 63. Specifications in our framework are declarative, i.e., they do not refer to the database internals and thus allow reasoning about the database behaviour at a higher abstraction level. To achieve this, we take an *axiomatic* approach similar to the one used to define the semantics of weak memory models of multiprocessors and shared-memory programming languages [3]: our specifications model database computations by *abstract executions*, which are structures of events and relations on them, reminiscent of event structures [26]. For example, Figure 3(a), page 63 gives an execution that could arise from the program (1). The boxes named $T_1$, $T_2$ and $T_3$ depict transactions, which are sequences of events ordered by the *program order* $\mathsf{po}$, reflecting the program syntax. The *visibility* edges $T_1 \xrightarrow{\mathsf{VIS}} T_2$ and $T_2 \xrightarrow{\mathsf{VIS}} T_3$ mean that the transaction $T_2$ ($T_3$) is aware of the updates made by $T_1$ ($T_2$). Consistency models are specified by *consistency axioms*, constraining abstract executions; e.g., a consistency axiom may require the visibility relation to be transitive and thereby disallow the execution in Figure 3(a).

The key observation we exploit in our framework is that modern consistency models for replicated databases usually guarantee *atomic visibility*: either all or none of the events in a transaction can be visible to another transaction; it is the flexibility in *when* a transaction becomes visible that leads to anomalies. Thanks to atomic visibility, in abstract executions we can use relations on whole transactions (such as $\mathsf{VIS}$ in Figure 3(a)), rather than on separate events inside them, thereby achieving particularly concise specifications. We further illustrate the benefits of this form of the specifications by exploiting it to obtain sufficient and necessary conditions for *observational refinement* [16] between transactions. This allows replacing a transaction in an abstract execution by another one without invalidating the consistency axioms of a given model. One can think of our conditions as characterising the optimisations that the database can soundly perform inside a transaction due to its atomic visibility.

To ensure that our declarative axiomatic specifications indeed faithfully describe the database behaviour, we prove that they are equivalent to alternative *operational* ones, given as algorithms closer to actual implementations (Theorem 6, §4). This correspondence also highlights implementation features that motivate the form of the consistency axioms.

Our work systematises the knowledge about consistency models of replicated databases and provides insights into relationships between them (§3). The proposed specification framework also gives a basis to develop methods for reasoning about application programs using weakly consistent databases. Finally, our framework is an effective tool for exploring the space of consistency models, because their concise axiomatic specifications allow easily experimenting with alternative designs. In particular, our formalisation naturally suggests a new consistency model (§3).

## 2 Abstract Executions

We consider a database storing *objects* $\mathsf{Obj} = \{x, y, \dots\}$, which for simplicity we assume to be integer-valued. Clients interact with the database by issuing `read` and `write` operations on the objects, grouped into *transactions*. We let $\mathsf{Op} = \{\mathtt{read}(x, n), \mathtt{write}(x, n) \mid x \in \mathsf{Obj}, n \in \mathbb{Z}\}$ describe the possible operation invocations: reading a value $n$ from an object $x$ or writing $n$ to $x$.

To specify a consistency model, we need to define the set of all client-database interactions that it allows. We start by introducing structures for recording such interactions in a single database computation, called *histories*. In these, we denote operation invocations using *history events* of the form $(\iota, o)$, where $\iota$ is an identifier from a countably infinite set $\mathsf{EventId}$ and $o \in \mathsf{Op}$. We use $e, f, g$ to range over history events. We let $\mathsf{WEvent}_x = \{(\iota, \mathtt{write}(x, n)) \mid \iota \in \mathsf{EventId}, n \in \mathbb{Z}\}$, define the set $\mathsf{REvent}_x$ of read events similarly, and let $\mathsf{HEvent}_x = \mathsf{REvent}_x \cup \mathsf{WEvent}_x$. A relation is a *total order* if it is transitive, irreflexive, and relates every two distinct elements one way or another.

▶ **Definition 1.** A *transaction* $T, S, \dots$ is a pair $(E, \mathsf{po})$, where $E \subseteq \mathsf{HEvent}$ is a finite, non-empty set of events with distinct identifiers, and the *program order* $\mathsf{po}$ is a total order over $E$. A *history* $\mathcal{H}$ is a (finite or infinite) set of transactions with disjoint sets of event identifiers.

All transactions in a history are assumed to be committed: to simplify presentation, our specifications do not constrain values read inside aborted or ongoing transactions.

To define the set of histories allowed by a given consistency model, we introduce *abstract executions*, which enrich histories with certain relations on transactions, declaratively describing how the database processes them. Consistency models are then defined by constraining these relations. We call a relation *prefix-finite*, if every element has finitely many predecessors in the transitive closure of the relation.

▶ **Definition 2.** An *abstract execution* is a triple $\mathcal{A} = (\mathcal{H}, \mathsf{VIS}, \mathsf{AR})$ where:

- *visibility* $\mathsf{VIS} \subseteq \mathcal{H} \times \mathcal{H}$ is a prefix-finite, acyclic relation; and
- *arbitration* $\mathsf{AR} \subseteq \mathcal{H} \times \mathcal{H}$ is a prefix-finite, total order such that $\mathsf{AR} \supseteq \mathsf{VIS}$.

We often write $T \xrightarrow{\mathsf{VIS}} S$ in lieu of $(T, S) \in \mathsf{VIS}$, and similarly for $\mathsf{AR}$. Figure 3(a) gives an execution corresponding to the anomaly explained in §1. Informally, $T \xrightarrow{\mathsf{VIS}} S$ means that $S$ is aware of $T$, and thus $T$'s effects can influence the results of operations in $S$. In implementation terms, this may be the case if the updates performed by $T$ have been delivered to the replica performing $S$; the prefix-finiteness requirement ensures that there may only be finitely many such transactions $T$. We call transactions unrelated by visibility *concurrent*. The relationship $T \xrightarrow{\mathsf{AR}} S$ means that the versions of objects written by $S$ supersede those written by $T$; e.g., *comment* supersedes *empty* in Figure 3(a). The constraint $\mathsf{AR} \supseteq \mathsf{VIS}$ ensures that writes by a transaction $T$ supersede those that $T$ is aware of; thus $\mathsf{AR}$ essentially

orders writes only by concurrent transactions. In an implementation, arbitration can be established by assigning timestamps to transactions.

A consistency model specification is a set of *consistency axioms* $\Phi$ constraining executions. The model allows those histories for which there exists an execution that satisfies the axioms:

$$\mathsf{Hist}_\Phi = \{\mathcal{H} \mid \exists \mathsf{VIS}, \mathsf{AR}.\, (\mathcal{H}, \mathsf{VIS}, \mathsf{AR}) \models \Phi\}. \tag{2}$$

Our consistency axioms do not restrict the operations done by the database clients. We can obtain the set of histories produced by a particular program interacting with the database, such as (1), by restricting the above set, as is standard in weak memory model definitions [7].

## 3 Specifying Transactional Consistency Models

We now apply the concepts introduced to define several existing consistency models; see Figures 1–3. For a total order $\mathsf{R}$ and a set $A$, we let $\max_{\mathsf{R}}(A)$ be the element $u \in A$ such that $\forall v \in A.\, v = u \vee (v, u) \in \mathsf{R}$; if $A = \emptyset$, then $\max_{\mathsf{R}}(A)$ is undefined. In the following, the use of $\max_{\mathsf{R}}(A)$ in an expression implicitly assumes that it is defined. For a relation $\mathsf{R} \subseteq A \times A$ and an element $u \in A$, we let $\mathsf{R}^{-1}(u) = \{v \mid (v, u) \in \mathsf{R}\}$. We denote the sequential composition of relations $\mathsf{R}_1$ and $\mathsf{R}_2$ by $\mathsf{R}_1 ; \mathsf{R}_2$. We write $_$ for a value that is irrelevant and implicitly existentially quantified.

**Baseline consistency model: Read Atomic.** The weakest consistency model we consider, Read Atomic (Figure 1), is defined by the axioms INT and EXT (Figure 2), which determine the outcomes of reads in terms of the visibility and arbitration relations. Consistency models stronger than Read Atomic are defined by adding axioms that constrain these relations. The *internal consistency axiom* INT ensures that, within a transaction, the database provides sequential semantics: a read from an object returns the same value as the last write to or read from this object in the transaction. In particular, INT guarantees that, if a transaction writes to an object and then reads the object, then it will observe its last write. The axiom also disallows so-called *unrepeatable reads*: if a transaction reads an object twice without writing to it in-between, it will read the same value in both cases.

If a read is not preceded in the program order by an operation on the same object, then its value is determined in terms of writes by other transactions using the *external consistency axiom* EXT. The formulation of EXT relies on the following notation, defining certain attributes of a transaction $T = (E, \mathsf{po})$. We let $T \vdash \mathsf{Write}\ x : n$ if $T$ writes to $x$ and the last value written is $n$: $\max_{\mathsf{po}}(E \cap \mathsf{WEvent}_x) = (_, \mathtt{write}(x, n))$. We let $T \vdash \mathsf{Read}\ x : n$ if $T$ makes an *external* read from $x$, i.e., one before writing to $x$, and $n$ is the value returned by the first such read: $\min_{\mathsf{po}}(E \cap \mathsf{HEvent}_x) = (_, \mathtt{read}(x, n))$. In this case, INT ensures that $n$ will be the result of all external reads from $x$ in $T$. According to EXT, the value returned by an external read in $T$ is determined by the transactions $\mathsf{VIS}$-preceding $T$ that write to $x$: if there are no such transactions, then $T$ reads the initial value 0; otherwise it reads the final value written by the last such transaction in $\mathsf{AR}$. (In examples we sometimes use initial values other than 0.) For example, the execution in Figure 3(a) satisfies EXT; if it included the edge $T_1 \xrightarrow{\mathsf{VIS}} T_3$, then EXT would force the read from $x$ in $T_3$ to return *post*. The axiom EXT implies the absence of so-called *dirty reads*: a committed transaction cannot read a value written by an aborted or an ongoing transaction (which are not present in abstract executions), and a transaction cannot read a value that was overwritten by the transaction that wrote it (ensured by the definition of $T \vdash \mathsf{Write}\ x : n$). Finally, EXT guarantees *atomic visibility* of a transaction: either all or none of its writes can be visible to another transaction.

For example, EXT disallows the execution in Figure 3(b) and, in fact, any execution with the same history. This illustrates a *fractured reads* anomaly: $T_1$ makes *Alice* and *Bob* friends, but $T_2$ observes only one direction of the friendship relationship. Thus, the consistency guarantees provided by Read Atomic are useful because they allow maintaining integrity invariants, such as the symmetry of the friendship relation.

**Stronger consistency models.** Even though Read Atomic ensures that all writes by a transaction become visible together, it does not constrain *when* this happens. This leads to a number of anomalies, including the causality violation shown in Figure 3(a). We now consider stronger consistency models that provide additional guarantees about the visibility of transactions. We specify the first model of *causal consistency* by requiring VIS to be transitive (TRANSVIS). This implies that transactions ordered by VIS (such as $T_1$ and $T_2$ in Figure 3(a)), are observed by others (such as $T_3$) in this order. Hence, the axiom TRANSVIS disallows the anomaly in Figure 3(a).

Both Read Atomic and causal consistency can be implemented without requiring any coordination among replicas [6, 19]: a replica can decide to commit a transaction without consulting other replicas. This allows the database to stay available even during network failures. However, the above consistency models allow the *lost update* anomaly illustrated by the execution in Figure 3(c), which satisfies the axioms of causal consistency. This execution could arise from the code, also shown in the figure, that uses transactions $T_1$ and $T_2$ to make deposits into an account. The two transactions read the initial balance of the account and concurrently modify it, resulting in one deposit getting lost. The next consistency model we consider, parallel snapshot isolation, prohibits such anomalies in exchange for requiring replica coordination in its implementations [24]. We specify it by strengthening causal consistency with the axiom NOCONFLICT, which does not allow transactions writing to the same object to be concurrent. This rules out any execution with the history in Figure 3(c): it forces $T_1$ and $T_2$ to be ordered by VIS, so that they cannot both read 0 from `acct`.

The axiom TRANSVIS in causal consistency and parallel snapshot isolation guarantees that VIS-ordered transactions are observed by others in this order (cf. Figure 3(a)). However, the axiom allows two *concurrent* transactions to be observed in different orders, as illustrated by the *long fork* anomaly in Figure 3(d), allowed by both models. Concurrent transactions $T_1$ and $T_2$ write to $x$ and $y$, respectively. A transaction $T_3$ observes the write to $x$, but not $y$, and a transaction $T_4$ observes the write to $y$, but not $x$. Thus, from the perspectives of $T_3$ and $T_4$, the writes of $T_1$ and $T_2$ happen in different orders.

The next pair of consistency models that we consider disallow this anomaly. We specify prefix consistency and snapshot isolation by strengthening causal consistency, respectively, parallel snapshot isolation, with the requirement that all transactions become visible throughout the system in the same order given by AR. This is formalised by the axiom PREFIX: if $T$ observes $S$, then it also observes all AR-predecessors of $S$. Since $\mathsf{AR} \supseteq \mathsf{VIS}$, PREFIX implies TRANSVIS. The axiom PREFIX disallows any execution with the history in Figure 3(d): $T_1$ and $T_2$ have to be related by AR one way or another; but then by PREFIX, either $T_4$ has to observe *post1* or $T_3$ has to observe *post2*.

Even though consistent prefix and snapshot isolation ensure that transactions become visible to others in the same order, they allow this to happen with a delay, caused by asynchronous propagation of updates in implementations. This leads to the *write skew* anomaly shown in Figure 3(e). Here each of $T_1$ and $T_2$ checks that the combined balance of two accounts exceeds 100 and, if so, withdraws 100 from one of them. Both transactions pass the checks and make the withdrawals from different accounts, resulting in the combined

| Φ | Consistency model | Axioms (Figure 2) | Fractured reads | Causality violation | Lost update | Long fork | Write skew |
|---|---|---|---|---|---|---|---|
| **RA** | Read Atomic [6] | INT, EXT | ✘ | ✓ | ✓ | ✓ | ✓ |
| **CC** | Causal consistency [19, 12] | INT, EXT, TRANSVIS | ✘ | ✘ | ✓ | ✓ | ✓ |
| **PSI** | Parallel snapshot isolation [24, 21] | INT, EXT, TRANSVIS, NOCONFLICT | ✘ | ✘ | ✘ | ✓ | ✓ |
| **PC** | Prefix consistency [13] | INT, EXT, PREFIX | ✘ | ✘ | ✓ | ✘ | ✓ |
| **SI** | Snapshot isolation [8] | INT, EXT, PREFIX, NOCONFLICT | ✘ | ✘ | ✘ | ✘ | ✓ |
| **SER** | Serialisability [20] | INT, EXT, TOTALVIS | ✘ | ✘ | ✘ | ✘ | ✘ |

**RA** ⊋ **CC** ⊋ **PC**, **PSI**; **PC** ⊋ **SI**; **PSI** ⊋ **SI**; **SI** ⊋ **SER**

**Figure 1** Consistency model definitions, anomalies and relationships.

$$\forall (E, \mathsf{po}) \in \mathcal{H}.\, \forall e \in E.\, \forall x, n.\, (e = (_, \mathtt{read}(x, n)) \wedge (\mathsf{po}^{-1}(e) \cap \mathsf{HEvent}_x \neq \emptyset)) \implies \max_{\mathsf{po}}(\mathsf{po}^{-1}(e) \cap \mathsf{HEvent}_x) = (_, _(x, n)) \qquad (\text{INT})$$

$$\forall T \in \mathcal{H}.\, \forall x, n.\, T \vdash \mathsf{Read}\ x : n \implies ((\mathsf{VIS}^{-1}(T) \cap \{S \mid S \vdash \mathsf{Write}\ x : _\} = \emptyset \wedge n = 0) \vee \max_{\mathsf{AR}}(\mathsf{VIS}^{-1}(T) \cap \{S \mid S \vdash \mathsf{Write}\ x : _\}) \vdash \mathsf{Write}\ x : n) \qquad (\text{EXT})$$

$\mathsf{VIS}$ is transitive (TRANSVIS) | $\mathsf{AR}; \mathsf{VIS} \subseteq \mathsf{VIS}$ (PREFIX) | $\mathsf{VIS}$ is total (TOTALVIS)

$$\forall T, S \in \mathcal{H}.\, (T \neq S \wedge T \vdash \mathsf{Write}\ x : _ \wedge S \vdash \mathsf{Write}\ x : _) \implies (T \xrightarrow{\mathsf{VIS}} S \vee S \xrightarrow{\mathsf{VIS}} T) \qquad (\text{NOCONFLICT})$$

**Figure 2** Consistency axioms, constraining an execution $(\mathcal{H}, \mathsf{VIS}, \mathsf{AR})$.

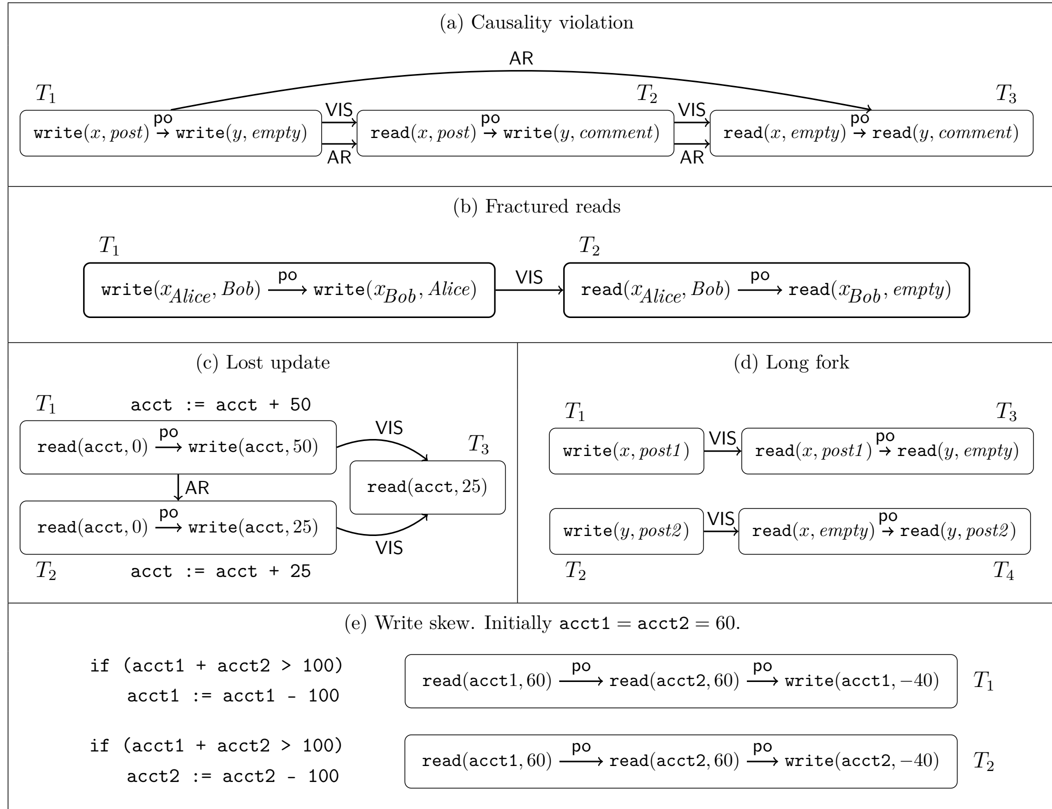


**Figure 3** Executions illustrating anomalies allowed by different consistency models. The boxes group events into transactions. We sometimes omit irrelevant AR edges.

balance going negative. NoConflict allows the transactions to be concurrent, because they write to different objects.

Write skew and all the other anomalies mentioned above are disallowed by the classical consistency model of serialisability. Informally, a history is serialisable if the results of operations in it could be obtained by executing its (committed) transactions in some total order according to the usual sequential semantics. We formalise this in our framework by the axiom TotalVis, which requires the visibility relation $\mathsf{VIS}$ to be total. Since we always have $\mathsf{AR} \supseteq \mathsf{VIS}$, it is easy to see that there is no execution with the history in Figure 3(e) and a total $\mathsf{VIS}$ that would satisfy Ext.

**Ramifications.** The above specifications demonstrate the benefits of using our framework. First, the specifications are *declarative*, since they state constraints on database processing in terms of $\mathsf{VIS}$ and $\mathsf{AR}$ relations, rather than the database internals. The specifications thus allow checking whether a consistency model admits a given history solely in terms of these relations, as per (2).

The declarative nature of our specifications also provides a *better understanding* of consistency models. In particular, it makes apparent the relationships between different models and highlights the main mechanism of strengthening consistency—mandating that more edges be included into visibility.

► **Proposition 3.** *The strict inclusions between the consistency models in Figure 1 hold.*

The strictness of the inclusions in Figure 1 follows from the examples of histories in Figure 3.

Axiomatic specifications also provide an effective tool for *designing new consistency models*. For example, the existing consistency models do not include a counterpart of Read Atomic obtained by adding the NoConflict axiom. Such an "Update Atomic" consistency model would prevent lost update anomalies without having to enforce causal consistency (as in parallel snapshot isolation), which incurs performance overheads [5]. Update Atomic could be particularly useful when *mixed* with Read Atomic, so that the NoConflict axiom apply only to some transactions specified by the programmer. This provides a lightweight way of strengthening consistency where necessary.

**Atomic visibility and observational refinement.** Our specifications are particularly concise because they are tailored to consistency models providing atomic visibility. With axioms Int and Ext establishing this property, additional guarantees can be specified while abstracting from the internal events in transactions: solely in terms of $\mathsf{VIS}$ and $\mathsf{AR}$ relations on whole transactions and transaction attributes given by the $\vdash$-judgements. To further illustrate the benefits of this way of specification, we now exploit it to establish sufficient and necessary conditions for when one transaction *observationally refines* another, i.e., we can replace it in an execution without invalidating the consistency axioms. This notion is inspired by that of testing preorders in process algebras [16]. We can think of it as characterising the optimisations that the database can soundly perform inside the transaction due to its atomic visibility. As it happens, the conditions we establish differ subtly depending on the consistency model.

To formulate observational refinement, we introduce ***contexts*** $\mathcal{X}$—abstract executions with a hole $[\,]$ that represents a transaction with an unspecified behaviour: $\mathcal{X} = (\mathcal{H} \cup \{[\,]\}, \mathsf{VIS}, \mathsf{AR})$, where $\mathsf{VIS}, \mathsf{AR} \subseteq (\mathcal{H} \cup \{[\,]\}) \times (\mathcal{H} \cup \{[\,]\})$ satisfy the conditions in Definition 2. We can fill in the hole in the above context $\mathcal{X}$ by a transaction $T$, provided that the sets of event identifiers appearing in $T$ and $\mathcal{H}$ are disjoint. This yields the abstract execution

$\mathcal{X}[T] = (\mathcal{H} \cup \{T\}, \mathsf{VIS}[[\,] \mapsto T], \mathsf{AR}[[\,] \mapsto T])$, where $\mathsf{VIS}[[\,] \mapsto T]$ treats $T$ in the same way as VIS treats $[\,]$ and similarly for $\mathsf{AR}[[\,] \mapsto T]$ (we omit the formal definition to conserve space). We say that a transaction $T_1$ ***observationally refines*** a transaction $T_2$ on the consistency model $\Phi$, written $T_1 \sqsubseteq_\Phi T_2$, if $\forall \mathcal{X}. \mathcal{X}[T_1] \models \Phi \implies \mathcal{X}[T_2] \models \Phi$.

▶ **Theorem 4.** *Let $T_1, T_2$ be such that $(\{T_1, T_2\}, \emptyset, \emptyset) \models$ INT. We have $T_1 \sqsubseteq_{\mathbf{RA}} T_2$ if and only if for all $x, n$:*

$$\big(\neg(T_1 \vdash \mathsf{Read}\ x : n) \implies \neg(T_2 \vdash \mathsf{Read}\ x : n)\big) \wedge \big(T_1 \vdash \mathsf{Write}\ x : n \iff T_2 \vdash \mathsf{Write}\ x : n\big).$$

*For $\Phi \in \{\mathbf{CC}, \mathbf{PC}, \mathbf{SER}\}$ we have $T_1 \sqsubseteq_\Phi T_2$ if and only if for all $x, n, m, l$:*

$$\big(\neg(T_1 \vdash \mathsf{Read}\ x : n) \implies (\neg(T_2 \vdash \mathsf{Read}\ x : n) \wedge (T_1 \vdash \mathsf{Write}\ x : n \iff T_2 \vdash \mathsf{Write}\ x : n))\big) \wedge$$
$$\big((T_1 \vdash \mathsf{Read}\ x : n \wedge (T_1 \vdash \mathsf{Write}\ x : m \implies m = n)) \implies (T_2 \vdash \mathsf{Write}\ x : l \implies l = n)\big).$$

*For $\Phi \in \{\mathbf{SI}, \mathbf{PSI}\}$ we have $T_1 \sqsubseteq_\Phi T_2$ if and only if $T_1 \sqsubseteq_{\mathbf{CC}} T_2$ and for all $x, n$:*

$$\neg(T_1 \vdash \mathsf{Write}\ x : n) \implies \neg(T_2 \vdash \mathsf{Write}\ x : n).$$

We prove the theorem in [14, §A]. In the case of $\Phi = \mathbf{RA}$, we prohibit $T_2$ from reading more objects than $T_1$ or changing the values read by $T_1$; however, it is safe for $T_2$ to read less than $T_1$. We also require $T_1$ and $T_2$ to have the same sets of final writes. The case of $\Phi \in \{\mathbf{CC}, \mathbf{PC}\}$ introduces two exception to the latter requirement. One exception is when $T_1$ reads an object and writes the same value to it. Then $T_2$ may not change the value written, but may omit the write. Another exception is when $T_1$ reads an object, but does not write to it. Then $T_2$ can write the value read without invalidating the reads in the context. This is disallowed when $\Phi \in \{\mathbf{SI}, \mathbf{PSI}\}$.

# 4 Operational Specifications

To justify that our axiomatic specifications of weak consistency models indeed faithfully describe the intended database behaviour, we now prove that they are equivalent to alternative *operational* ones. These are given as algorithms that are close to actual implementations [6, 19, 13, 24], yet abstract from some of the more low-level features that such implementations have. We start by giving an operational specification of the weakest consistency model we consider, Read Atomic. We then specify other models weaker than serialisability by assuming additional guarantees about the communication between replicas in this algorithm.

## 4.1 Operational Specification of Read Atomic

Informally, the idealised algorithm for Read Atomic operates as follows. The database consists of a set of *replicas*, identified by $\mathsf{RId} = \{r_0, r_1, \ldots\}$, each maintaining a copy of all objects. The set RId is infinite, to model dynamic replica creation. We assume that the system is fully connected: each replica can broadcast messages to all others. All client operations within a given transaction are initially executed at a single replica (though operations in *different* transactions can be executed at different replicas). For simplicity, we assume that every transaction eventually terminates. When this happens, the replica decides whether to commit or abort it. In the former case, the replica sends a message to all other replicas containing the *transaction log*, which describes the updates done by the transaction. The replicas incorporate the updates into their state upon receiving the message. A transaction log has the form $t : \rho$, where $\rho \in \{\mathtt{write}(x, n) \mid x \in \mathsf{Obj}, n \in \mathbb{N}\}^* \triangleq \mathsf{UpdateList}$. This gives

the sequence of values written to objects and the unique *timestamp* $t \in \mathbb{N}$ of the transaction, which is used to determine the precedence of different object versions (and thus implements the AR relation in abstract executions). We denote the set of all sets of logs with distinct timestamps by LogSet.

Every replica processes transactions locally without interleaving. This idealisation does not limit generality, since all anomalies that would result from concurrent execution of transactions at a single replica arise anyway because of the asynchronous propagation of updates between replicas. The above assumption allows us to maintain the state of a replica $r$ in the algorithm by a pair $(D, l) \in \mathsf{RState} \triangleq \mathsf{LogSet} \times (\mathsf{UpdateList} \uplus \{\mathsf{idle}\})$, where:

- $l$ is either the sequence of updates done so far by the (single) transaction currently executing at $r$, or idle, signifying that no transaction is currently executing; and
- $D$ is the database copy of $r$, represented by the set of logs of transactions that have committed at $r$ or have been received from other replicas.

Then a *configuration* of the whole system $(R, M) \in \mathsf{Config} \triangleq (\mathsf{RId} \to \mathsf{RState}) \times \mathsf{LogSet}$ is described by the state $R(r)$ of every replica $r$ and the pool of messages $M$ in transit among replicas.

Formally, our algorithm is defined using the transition relation $\twoheadrightarrow$: $\mathsf{Config} \times \mathsf{LEvent} \times \mathsf{Config}$ in Figure 4, which describes how system configurations change in response to *low-level events* from a set LEvent, describing actions by clients and message receipts by replicas. The set LEvent consists of triples of the form $(\iota, r, \mathbf{o})$, where $\iota \in \mathsf{EventId}$ is the event identifier, $r \in \mathsf{RId}$ is the replica the event occurs at, and $\mathbf{o}$ is a *low-level operation* from the set

$$\mathsf{COp} = \{\mathtt{start}, \mathtt{read}(x, n), \mathtt{write}(x, n), \mathtt{commit}(t), \mathtt{abort}, \mathtt{receive}(t : \rho) \mid x \in \mathsf{Obj}, n \in \mathbb{Z}, t \in \mathbb{N}, \rho \in \mathsf{UpdateList}\}.$$

We use $\mathbf{e}, \mathbf{f}, \mathbf{g}$ to range over low-level events.

According to $\twoheadrightarrow$, when a client starts a transaction at a replica $r$ (Start), the database initialises the current sequence of updates to signify that a transaction is in progress. Since a replica processes transactions serially, a transaction can start only if $r$ is not already executing a transaction. When a client writes $n$ to an object $x$ at a replica $r$ (Write), the corresponding record $\mathtt{write}(x, n)$ is appended to the current sequence of updates. This rule can be applied only when $r$ is not executing a transaction. A read of an object $x$ at $r$ (Read) returns the value determined by a lastval function based on the transactions in $r$'s database copy and the current transaction. For $D' \in \mathsf{LogSet}$ we define $\mathsf{lastval}(x, D')$ as the last value written to $x$ by the transaction with the highest timestamp among those in $D'$, or 0 if $x$ is not mentioned in $D'$. Since the timestamps of transactions in $D'$ are distinct, this defines $\mathsf{lastval}(x, D')$ uniquely. For brevity, we omit its formal definition. Note that (Read) implies that a transaction always reads from its own writes and a snapshot of the database the replica had at its start; the transaction is not affected by writes concurrently executing at other replicas, thus ensuring the absence of unrepeatable reads (§3).

If a transaction aborts at a replica $r$ (Abort), the current sequence of updates of $r$ is cleared. If the transaction commits (Commit), it gets assigned a timestamp $t$, and its log is added to the message pool, as well as to $r$'s database copy. The timestamp $t$ is chosen to be greater than the timestamps of all the transactions in $r$'s database copy, which validates the condition $\mathsf{AR} \supseteq \mathsf{VIS}$ in Definition 2. The timestamp $t$ also has to be distinct from any timestamp assigned previously in the execution. The fact that (Commit) sends all updates by a transaction in a single message ensures atomic visibility. Note that, in Read Atomic, a transaction can always commit; as we explain in the following, this is not the case for some

$$\text{(Start)} \quad \frac{\mathbf{e} = (_, r, \texttt{start})}{(R[r \mapsto (D, \mathsf{idle})], M) \xrightarrow{\mathbf{e}} (R[r \mapsto (D, \varepsilon)], M)}$$

$$\text{(Write)} \quad \frac{\mathbf{e} = (_, r, \texttt{write}(x, n))}{(R[r \mapsto (D, \rho)], M) \xrightarrow{\mathbf{e}} (R[r \mapsto (D, \rho \cdot \texttt{write}(x, n))], M)}$$

$$\text{(Read)} \quad \frac{\mathbf{e} = (_, r, \texttt{read}(x, n)) \qquad n = \mathsf{lastval}(x, D \cup \{\infty : \rho\})}{(R[r \mapsto (D, \rho)], M) \xrightarrow{\mathbf{e}} (R[r \mapsto (D, \rho)], M)}$$

$$\text{(Abort)} \quad \frac{\mathbf{e} = (_, r, \texttt{abort})}{(R[r \mapsto (D, \rho)], M) \xrightarrow{\mathbf{e}} (R[r \mapsto (D, \mathsf{idle})], M)}$$

$$\text{(Commit)} \quad \frac{\begin{array}{l}\mathbf{e} = (_, r, \texttt{commit}(t)) \\ (\forall r', D'.\, R(r') = (D', _) \implies (t : _) \notin D') \qquad (\forall t'.\, (t' : _) \in D \implies t > t')\end{array}}{(R[r \mapsto (D, \rho)], M) \xrightarrow{\mathbf{e}} (R[r \mapsto (D \cup \{t : \rho\}, \mathsf{idle})], M \cup \{t : \rho\})}$$

$$\text{(Receive)} \quad \frac{\mathbf{e} = (_, r, \texttt{receive}(t : \rho))}{(R[r \mapsto (D, \mathsf{idle})], M \cup \{(t : \rho)\}) \xrightarrow{\mathbf{e}} (R[r \mapsto (D \cup \{(t : \rho)\}, \mathsf{idle})], M \cup \{t : \rho\})}$$

**Figure 4** Transition relation $\twoheadrightarrow$: $\mathsf{Config} \times \mathsf{LEvent} \times \mathsf{Config}$ for defining the operational specification. We let $R[r \mapsto u]$ be the function that has the same value as $R$ everywhere except $r$, where it has the value $u$; $\cdot$ denotes sequence concatenation, and $\varepsilon$ the empty sequence.

$$\big(\mathbf{e}_1 \in \{(_, r, \texttt{receive}(t_1 : _)), (_, r, \texttt{commit}(t_1))\} \land \mathbf{e}_2 = (_, r, \texttt{commit}(t_2)) \land \mathbf{e}_1 \prec \mathbf{e}_2 \land r \neq r' \land \mathbf{f}_2 = (_, r', \texttt{receive}(t_2 : _))\big) \implies \big(\exists \mathbf{f}_1 \in \{(_, r', \texttt{receive}(t_1 : _)), (_, r', \texttt{commit}(t_1))\}.\, \mathbf{f}_1 \prec \mathbf{f}_2\big) \quad \text{(CausalDeliv)}$$

$$\big(\mathbf{e}_1 = (_, _, \texttt{commit}(t_1)) \land \mathbf{e}_2 = (_, _, \texttt{commit}(t_2)) \land \mathbf{e}_1 \prec \mathbf{e}_2\big) \implies t_1 < t_2 \quad \text{(MonTS)}$$

$$\big(\mathbf{g} = (_, r, \texttt{start}) \land \mathbf{e}_2 \in \{(_, r, \texttt{commit}(t_2)), (_, r, \texttt{receive}(t_2 : _))\} \land \mathbf{f} = (_, _, \texttt{commit}(t_1)) \land t_1 < t_2 \land \mathbf{e}_2 \prec \mathbf{g}\big) \implies \big(\exists \mathbf{e}_1 \in \{(_, r, \texttt{commit}(t_1)), (_, r, \texttt{receive}(t_1 : _))\}.\, \mathbf{e}_1 \prec \mathbf{g}\big) \quad \text{(TotalDeliv)}$$

$$\big(\mathbf{e}_1 = (_, r, \texttt{write}(x, _)) \land \mathbf{f}_1 = (_, r, \texttt{commit}(t_1)) \land \mathsf{TS}_{\mathcal{C}}(\mathbf{e}_1) = t_1 \land \mathbf{e}_2 = (_, r', \texttt{write}(x, _)) \land \mathbf{f}_2 = (_, r', \texttt{commit}(t_2)) \land \mathsf{TS}_{\mathcal{C}}(\mathbf{e}_2) = t_2 \land \mathbf{f}_2 \prec \mathbf{f}_1 \land r \neq r'\big) \implies \big(\exists \mathbf{g} \in \mathbf{E}.\, \mathbf{g} = (_, r, \texttt{receive}(t_2 : _)) \land \mathbf{g} \prec \mathbf{f}_1\big), \quad \text{(ConflictCheck)}$$

where for $\mathbf{e} \in \mathbf{E}$ we let

$$\mathsf{TS}_{\mathcal{C}}(\mathbf{e}) = \begin{cases} t, & \text{if } \exists r.\, \mathbf{e} \in \{(_, r, \texttt{read}(_, _)), (_, r, \texttt{write}(_, _))\} \land \\ & \quad \exists \mathbf{g} \in \mathbf{E}.\, \mathbf{g} = (_, r, \texttt{commit}(t)) \land \\ & \quad \neg(\exists \mathbf{f} \in \{(_, r, \texttt{commit}(_)), (_, r, \texttt{abort})\}.\, (\mathbf{e} \prec \mathbf{f} \prec \mathbf{g})) \\ \text{undefined}, & \text{otherwise} \end{cases}$$

| $\Phi$ | Constraints | $\Phi$ | Constraints | $\Phi$ | Constraints |
|---|---|---|---|---|---|
| **RA** | None | **PSI** | (CausalDeliv), (ConflictCheck) | **SI** | (MonTS), (TotalDeliv), (ConflictCheck) |
| **CC** | (CausalDeliv) | **PC** | (MonTS), (TotalDeliv) | | |

**Figure 5** Constraints on concrete executions $\mathcal{C} = (\mathbf{E}, \prec)$ required by various consistency models. Free variables are universally quantified and range over the following domains: $\mathbf{e}_i, \mathbf{f}, \mathbf{f}_i, \mathbf{g} \in \mathbf{E}$ for $i = 1, 2$; $t_1, t_2 \in \mathbb{N}$; $r, r' \in \mathsf{RId}$.

of the other consistency models. Finally, a replica $r$ that is not executing a transaction can receive a transaction log from the message pool (Receive), adding it to the database copy.

We define the semantics of Read Atomic by considering all sequences of transitions generated by $\twoheadrightarrow$ from an initial configuration where the log sets of all replicas and the message pool are empty. We thereby consider all possible operations that clients could issue to the database.

▶ **Definition 5.** Let $(R_0, M_0) = (\lambda r. (\emptyset, \mathsf{idle}), \emptyset)$. A *concrete execution* is a pair $\mathcal{C} = (\mathbf{E}, \prec)$, where: $\mathbf{E} \subseteq \mathsf{LEvent}$; $\prec$ is a prefix-finite, total order on $\mathbf{E}$; and if $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, \dots$ is the enumeration of the events in $\mathbf{E}$ defined by $\prec$, then for some configurations $(R_1, M_1), (R_2, M_2), \dots \in$ $\mathsf{Config}$ we have $(R_0, M_0) \xrightarrow{\mathbf{e}_1} (R_1, M_1) \xrightarrow{\mathbf{e}_2} (R_2, M_2) \xrightarrow{\mathbf{e}_3} \dots$

## 4.2 Correspondence to Axiomatic Specifications and Other Models

We next show that the above operational specification indeed defines the semantics of Read Atomic, and that stronger models can be defined by assuming additional guarantees about communication between replicas. These guarantees are formalised by the constraints on concrete executions in Figure 5; in implementations they would be ensured by distributed protocols that our specifications abstract from.

We first map each concrete execution into a history, which includes only reads and writes in its committed transactions. The history of $\mathcal{C} = (\mathbf{E}, \prec)$ is defined as follows:

$$\begin{aligned} &\mathsf{history}(\mathcal{C}) = \big\{T_t \mid \{\mathbf{e} \in \mathbf{E} \mid \mathsf{TS}_{\mathcal{C}}(\mathbf{e}) = t\} \neq \emptyset\big\}, \text{ where } T_t = (E_t, \mathsf{po}_t) \text{ for} \\ &E_t = \{(\iota, \mathbf{o}) \mid \exists \mathbf{e} \in \mathbf{E}.\, \mathbf{e} = (\iota, _, \mathbf{o}) \land \mathsf{TS}_{\mathcal{C}}(\mathbf{e}) = t\}; \\ &\mathsf{po}_t = \{(\iota_1, \mathbf{o}_1), (\iota_2, \mathbf{o}_2) \mid (\iota_1, \mathbf{o}_1), (\iota_2, \mathbf{o}_2) \in E_t \land (\iota_1, _, \mathbf{o}_1) \prec (\iota_2, _, \mathbf{o}_2)\}, \end{aligned}$$

where $\mathsf{TS}_{\mathcal{C}}$ is defined in Figure 5. We lift the function $\mathsf{history}$ to sets of concrete executions as expected.

▶ **Theorem 6.** *For a consistency model $\Phi$ let $\mathsf{ConcExec}_\Phi$ be the set of concrete executions satisfying the model-specific constraints in Figure 5. Then $\mathsf{history}(\mathsf{ConcExec}_\Phi) = \mathsf{Hist}_\Phi$.*

**Proof outline.** We defer the full proof to [14, §B]. Here we sketch the argument for one set inclusion ($\subseteq$) and, on the way, explain the constraints in Figure 5. Fix a $\Phi$ and let $\mathcal{C} = (\mathbf{E}, \prec) \in \mathsf{ConcExec}_\Phi$. To show $\mathsf{history}(\mathcal{C}) \in \mathsf{Hist}_\Phi$ we let $\mathcal{A} = (\mathsf{history}(\mathcal{C}), \mathsf{VIS}, \mathsf{AR})$, where $\mathsf{AR} = \{(T_{t_1}, T_{t_2}) \mid t_1 < t_2\}$ and

$$\begin{aligned} \mathsf{VIS} = \big\{(T_{t_1}, T_{t_2}) \mid \exists \mathbf{e}_1, \mathbf{e}_2 \in \mathbf{E}.\, \exists r.\, \mathbf{e}_1 \in \{(_, r, \mathtt{commit}(t_1)), (_, r, \mathtt{receive}(t_1 : _))\} \land \\ \mathbf{e}_2 = (_, r, \mathtt{commit}(t_2)) \land \mathbf{e}_1 \prec \mathbf{e}_2\big\}. \end{aligned}$$

While $\mathsf{AR}$ merely lifts the order on timestamps to transactions, $\mathsf{VIS}$ reflects message delivery: $T_{t_1} \xrightarrow{\mathsf{VIS}} T_{t_2}$ if the effects of $T_{t_1}$ have been incorporated into the state of the replica where $T_{t_2}$ is executed. We can show that any abstraction execution $\mathcal{A}$ constructed from a concrete execution $\mathcal{C}$ as above satisfies INT and EXT, and hence, its history belongs to $\mathsf{Hist}_{\mathbf{RA}}$. The constraints on a concrete execution $\mathcal{C}$ in Figure 5 ensure that the abstract execution $\mathcal{A}$ constructed from it satisfies other axioms in Figure 2.

Constraint (CausalDeliv) implies the axiom TRANSVIS, because it ensures that the message delivery is *causal* [9]: if a replica $r$ sends the log of a transaction $t_2$ (event $\mathbf{e}_2$) after it sends or receives the log of $t_1$ (event $\mathbf{e}_1$), then every other replica $r'$ will receive the log of $t_2$ (event $\mathbf{f}_2$) only after it receives or sends the log of $t_1$ (event $\mathbf{f}_1$).

The axiom Prefix follows from constraints (MonTS) and (TotalDeliv). The constraint (MonTS) requires that timestamps agree with the order in which transactions commit. The constraint (TotalDeliv) requires that each transaction access a database snapshot that is closed under adding transactions with timestamps ($t_1$) smaller than the ones already present in the snapshot ($t_2$). In an implementation, the above constraints can be satisfied if replicas communicate via a central server, which assigns timestamps to transactions when they commit, and propagates their logs to replicas in the order of their timestamps [13].

The axiom NoConflict follows from the constraint (ConflictCheck), similar to that in the original definitions of **SI** [8] and **PSI** [24]. The constraint allows a transaction $t_1$ to commit at a replica $r$ (event $\mathbf{f}_1$) only if it passes a *conflict detection check*: if $t_1$ updates an object $x$ (event $\mathbf{e}_1$) that is also updated by a transaction $t_2$ (event $\mathbf{e}_2$) committed at another replica $r'$ (event $\mathbf{f}_2$), then the replica $r$ must have received the log of $t_2$ (event $\mathbf{g}$). If this check fails, the only option left for the database is to abort $t$ using the rule (Abort). Implementing the check in a realistic system would require the replica $r$ to coordinate with others on commit [24]. ◀

The above operational specifications are closer to the intuition of practitioners [6, 19, 13, 24] and thus serve to validate our axiomatic specifications. However, they are more verbose and reasoning about database behaviour using them may get unwieldy. It requires us to keep track of low-level information about the system state, such as the logs at all replicas and the set of messages in transit. We then need to reason about how the system state is affected by a large number of possible interleavings of operations at different replicas. In contrast, our axiomatic specifications (§3) are more declarative and, in particular, do not refer to implementation-level details, such as message exchanges between replicas. These specifications thereby facilitate reasoning about the database behaviour.

## 5 Related Work

Our specification framework builds on the axiomatic approach to specifying consistency models, previously applied to weak shared-memory models [3] and eventual consistency [11, 12]. In particular, the visibility and arbitration relations were first introduced for specifying eventual consistency and causally consistent transactions [12]. In comparison to prior work, we handle more sophisticated transactional consistency models. Furthermore, our framework is specifically tailored to transactional models with atomic visibility, by defining visibility and arbitration relations on whole transactions as opposed to events. This avoids the need to enforce atomic visibility explicitly in all axioms [12], thus simplifying specifications.

Adya [2] has previously proposed specifications for weak consistency models of transactions in classical databases. His framework also broadly follows the axiomatic specification approach, but uses relations different from visibility and arbitration. Adya's work did not address the variety of consistency models for large-scale databases proposed recently, while our framework is particularly appropriate for these. On the other hand, Adya handled transactional consistency models that do not guarantee atomic visibility, such as Read Committed, which we do not address. Adya also specified snapshot isolation (**SI**), which is a weak consistency model older than the others we consider. However, his specification is low-level, since it introduces additional events to denote the times at which a transaction takes a snapshot of the database state. Saeida Ardekani et al. [22] have since proposed a higher-level specification for snapshot isolation; this specification still uses relations on individual events and thus does not exploit atomic visibility.

Partial orders have been used to define semantics of concurrent and distributed programs, e.g., by event structures [26]. Our results extend this research line by considering new kinds of relations among events, appropriate to describe computations of weakly consistent databases, and by relating the resulting abstract specifications to lower-level algorithms.

Prior work has investigated calculi with transactions communicating via message passing: cJoin [10], $\mathsf{TCCS}^m$ [17] and RCCS [15]. Even though replicated database implementations and our operational specifications are also based on message passing, the database interface that we consider allows client programs only to read and write objects. Thereby, it provides the programs with an (imperfect) illusion of *shared memory*, and our goal was to provide specifications for this interface that abstract from its message passing-based implementation.

## 6 Conclusion

We have proposed a framework for uniformly specifying transactional consistency models of modern replicated databases. The axiomatic nature of our framework makes specifications declarative and concise, with further simplicity brought by exploiting atomic visibility. We have illustrated the use of the framework by specifying several existing consistency models and thereby systematising the knowledge about them. We have also validated our axiomatic specifications by proving their equivalence to operational specifications that are closer to implementations.

We hope that our work will promote an exchange of ideas between the research communities of large-scale databases and concurrency theory. In particular, our framework provides a basis to develop techniques for reasoning about the correctness of application programs using modern databases; this is the subject of our ongoing work.

Finally, axiomatic specifications are well-suited for systematically exploring the design space of consistency models. In particular, insights provided by the specifications may suggest new models, as we illustrated by the Update Atomic model in §3. This is likely to help in the design of the sophisticated programming interfaces that replicated databases are starting to provide to compensate for the weakness of their consistency models. For example, so-called replicated data types [23] avoid lost updates by eventually merging concurrent updates without coordination between replicas, and sessions [25] provide additional consistency guarantees for transactions issued by the same client. Finally, there are also interfaces that allow the programmer to request different consistency models for different transactions [18], analogous to fences in weak memory models [3]. In the future we plan to generalise our techniques to handle the above features. We expect to handle replicated data types by integrating our framework with their specifications proposed in [11], and to handle sessions and mixed consistency models by studying additional constraints on the visibility and arbitration relations. We believe that the complexity of database consistency models and the above programming interfaces makes it indispensable to specify them formally and declaratively. Our work provides the necessary foundation for achieving this.

**Acknowledgements.** We thank Artem Khyzha, Vasileios Koutavas, Hongseok Yang and the anonymous reviewers for comments that helped improve the paper. This work was supported by the EU FET project ADVENT.

## References

1 Daniel Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2), 2012.

2 Atul Adya. Weak consistency: A generalized theory and optimistic implementations for distributed transactions. PhD thesis, MIT, 1999.
3 Jade Alglave. A formal hierarchy of weak memory models. *FMSD*, 41(2), 2012.
4 Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly Available Transactions: virtues and limitations. In *VLDB*, 2014.
5 Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *SOCC*, 2012.
6 Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Scalable atomic visibility with RAMP transactions. In *SIGMOD*, 2014.
7 Mark Batty, Mike Dodds, and Alexey Gotsman. Library abstraction for C/C++ concurrency. In *POPL*, 2013.
8 Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.
9 Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1), 1987.
10 Roberto Bruni, Hernán C. Melgratti, and Ugo Montanari. cJoin: Join with communicating transactions. *Mathematical Structures in Computer Science*, 25(3), 2015.
11 Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In *POPL*, 2014.
12 Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. Eventually consistent transactions. In *ESOP*, 2012.
13 Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In *ECOOP*, 2015.
14 Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A framework for transactional consistency models with atomic visibility (extended version). Available from `http://software.imdea.org/~gotsman/`.
15 Vincent Danos and Jean Krivine. Transactions in RCCS. In *CONCUR*, 2005.
16 Rocco De Nicola and Matthew Hennessy. Testing equivalence for processes. In *ICALP*, 1983.
17 Vasileios Koutavas, Carlo Spaccasassi, and Matthew Hennessy. Bisimulations for communicating transactions (extended abstract). In *FOSSACS*, 2014.
18 Cheng Li, Daniel Porto, Allen Clement, Rodrigo Rodrigues, Nuno Preguiça, and Johannes Gehrke. Making geo-replicated systems fast if possible, consistent when necessary. In *OSDI*, 2012.
19 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.
20 Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4), 1979.
21 M. Saeida Ardekani, P. Sutra, and M. Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *SRDS*, 2013.
22 M. Saeida Ardekani, P. Sutra, M. Shapiro, and N. Preguiça. On the scalability of snapshot isolation. In *Euro-Par*, 2013.
23 Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *SSS*, 2011.
24 Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.
25 Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.
26 Glynn Winskel. Event structure semantics for CCS and related languages. In *ICALP*, 1982.